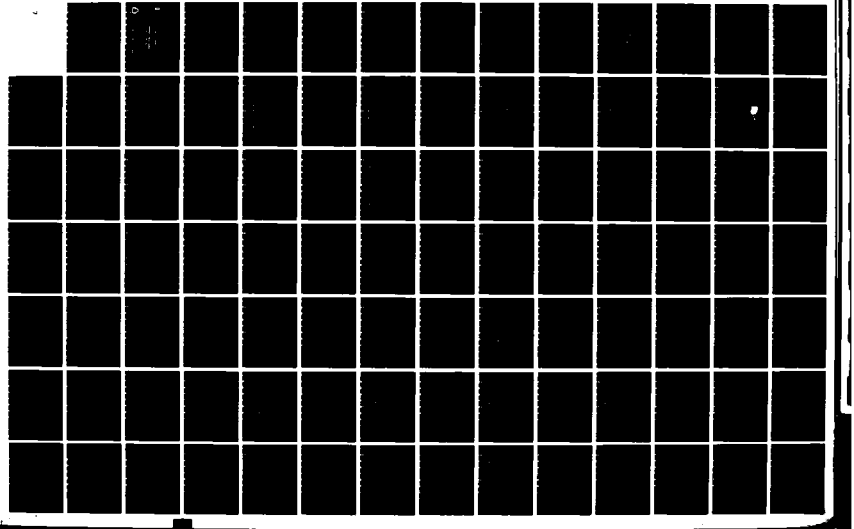
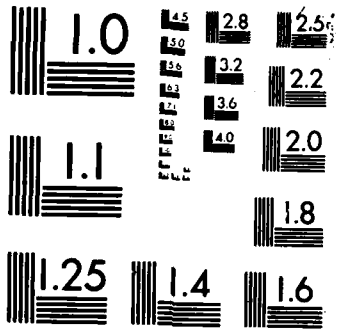
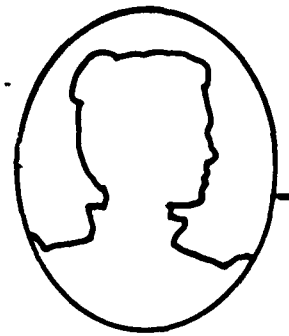
AD-A141 848 INTRODUCTION TO ADA A HIGHER ORDER LANGUAGE L103 TEACHER'S GUIDE(U) SOFTECH INC WALTHAM MA MAY 84 DAAB07-83-C-K514 1/4

UNCLASSIFIED F/G 9/2 NL







Ada® Training Curriculum

MAY 1984

Introduction to Ada®
A Higher Order Language
L103
Teacher's Guide

Center For Tactical Computer Systems
(CENTACS)

U.S. Army Communications-Electronics Command
(CECOM)

Contract DAAB07-83-C-K514

Prepared By:

SOFTECH, INC.
460 Totten Pond Road
Waltham, MA 02154

•Ada is a registered trademark of the U.S. Government, Ada Joint Program Office

*Approved For Public Release/Distribution Unlimited

INSTRUCTOR NOTES

THIS SECTION SHOULD TAKE 20 MINUTES OF LECTURE. BE SURE TO INTRODUCE YOURSELF AND HAVE THE STUDENTS INTRODUCE THEMSELVES AND WHAT THEY DO. DON'T SPEND A LOT OF TIME HERE.

SECTION 1
PREFACE

INSTRUCTOR NOTES

THESE ARE THE OBJECTIVES OF L103 (RATHER THAN OF HOL's).

NOTE LAST POINT ABOUT "WHY BOTHER?" ESPECIALLY. AN AUDIENCE OF EXPERIENCED ASSEMBLY LANGUAGE PROGRAMMERS IS LIKELY TO BE SKEPTICAL.

AVOID "PROPAGANDA" ABOUT HOL'S, AND SWEEPING GENERALIZATIONS THAT CAN'T BE BACKED UP WITH HARD FACTS. STICK TO SUPPORTABLE OBJECTIVE COMPARISONS.

OBJECTIVES

- INTRODUCE THE ASSEMBLY LANGUAGE PROGRAMMER TO HIGH-ORDER LANGUAGES
- PROVIDE A BROAD OVERVIEW OF FEATURES ("BIG PICTURE" -- SAVE DETAILS FOR LATER MODULES).
- ADDRESS THE FOLLOWING QUESTIONS:
  - WHAT DO WE MEAN BY HIGH-ORDER LANGUAGES?
  - WHAT DO HIGH-ORDER LANGUAGE PROGRAMS LOOK LIKE?
  - WHAT CAN THEY DO?
  - WHY BOTHER? WHY NOT USE ASSEMBLY LANGUAGE?
  - HOW IS PROGRAMMING IN A HOL DIFFERENT THAN IN ASSEMBLY?
- PRESENT AN INTRODUCTION TO A SPECIFIC HOL - ADA.

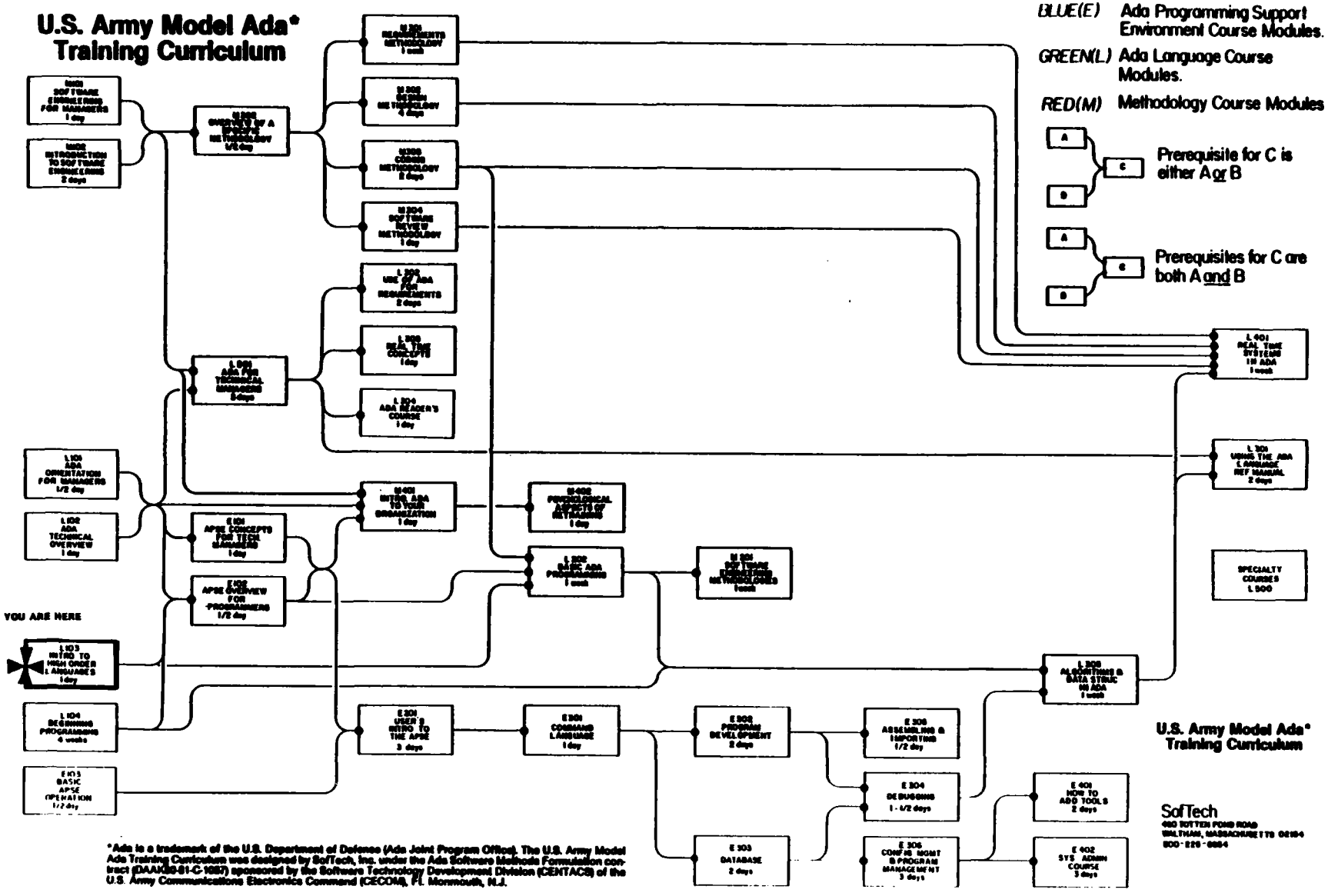
INSTRUCTOR NOTES

WHAT WILL NOT BE COVERED

- FUNDAMENTALS OF COMPUTER PROGRAMMING (EXCEPT WHERE DIFFERENT FOR HIGH-ORDER LANGUAGES)
- "NUTS AND BOLTS" OF SYNTAX
- ISSUES OF "STRONGLY NON-PROCEDURAL" LANGUAGES SUCH AS LISP AND APL (COURSE IS CENTERED AROUND ADA)

BE SURE TO POINT OUT WHERE THEY ARE IN THE CURRICULUM.

POINT OUT THAT TYPICALLY THEY WOULD HAVE PREVIOUSLY HAD M102, AND FROM HERE THEY WILL PROCEED TO L202 AND M303.


U.S. Army Model Ada* Training Curriculum
BLUE(E) Ada Programming Support Environment Course Modules.
GREEN(L) Ada Language Course Modules.
RED(M) Methodology Course Modules
Prerequisite for C is either A or B
Prerequisites for C are both A and B
YOU ARE HERE
U.S. Army Model Ada* Training Curriculum
SofTech
*Ada is a trademark of the U.S. Department of Defense (Ada Joint Program Office). The U.S. Army Model Ada Training Curriculum was designed by SofTech, Inc. under the Ada Software Methods Formulation contract (DAAK80-81-C-1087) sponsored by the Software Technology Development Division (CENTACS) of the U.S. Army Communications Electronics Command (CECOM), Ft. Monmouth, N.J.

POINT OUT THAT STUDENTS TO WHOM THESE ASSUMPTIONS DO NOT APPLY ARE IN THE WRONG COURSE. THEY BELONG IN L102.

ASSUMPTIONS

- ALL STUDENTS HAVE EXPERIENCE IN ASSEMBLY LANGUAGE PROGRAMMING
- STUDENTS HAVE LITTLE OR NO EXPERIENCE WITH HIGH-OI 'E .. .. 'ES
- STUDENTS EXPECT TO BE USING A HIGH-ORDER LANGUAGE, PROBABLY ADA, IN FUTURE WORK.

INSTRUCTOR NOTES

1. REGARDING DISCUSSIONS OF IMPLEMENTATIONS: SINCE THESE QUESTIONS WILL ARISE ANYWAY, GIVEN THE NATURAL CURIOSITY OF THE INTENDED AUDIENCE, IT IS PRESUMABLY BETTER TO DISPENSE WITH THEM DELIBERATELY, LEAST THEY LINGER AND DISTRACT THE CLASS FROM THE REAL ISSUES OF THE MODULE.

2. THIS IS A GOOD TIME TO MAKE THE POINT THAT QUESTIONS ARE ENCOURAGED THROUGHOUT THE MODULE. MENTION THAT SOME QUESTIONS WHICH SEEM SIMPLE ARE IN FACT ELABORATE AND TIME-CONSUMING (E.G., THEY ADDRESS ISSUES TO BE COVERED LATER IN THE COURSE).

3. IT IS IMPORTANT TO SET THE PROPER TONE. "WE ARE HERE TO PRESENT FACTS ABOUT HOLs, NOT TO DISCUSS WHY THINGS ARE THE WAY THEY ARE." DON'T LET MORALISTIC DISCUSSIONS GET OUT OF HAND.

HOW THIS MODULE WILL BE PRESENTED

1. INTRODUCE ELEMENTS AND FEATURES OF A MODERN HIGH-ORDER LANGUAGE, ADA.

   - WHY EACH FEATURE EXISTS

   - HOW IT IS USED

   - HOW IT COMPARES WITH SIMILAR ASSEMBLY LANGUAGE CONSTRUCTS, IF ANY

   - HOW IT MIGHT BE IMPLEMENTED -- BRIEF, INTUITIVE-LEVEL DISCUSSION OF SOME FEATURES.

INSTRUCTOR NOTES

HOW THIS MODULE WILL BE PRESENTED (Continued)

2. HIGHLIGHT SALIENT DIFFERENCES BETWEEN HIGH-ORDER LANGUAGES AND ASSEMBLY LANGUAGE.

- LANGUAGE STRUCTURE AND SEMANTICS
- DATA STRUCTURE
- DESIGN METHODOLOGY
- PERFORMANCE CRITERIA

INSTRUCTOR NOTES

HOW THIS MODULE WILL BE PRESENTED (Continued)

3. PRESENT THE HIGH-ORDER LANGUAGE VIEWED AS THE MACHINE LANGUAGE OF A HYPOTHETICAL, ABSTRACT MACHINE.

   PROGRAMMING FOR THE ABSTRACT MACHINE (RESISTING THE TEMPTATION TO PROGRAM FOR THE "REAL" MACHINE).

4. EMPHASIZE "TOP-DOWN" SOFTWARE DESIGN AND STRUCTURED PROGRAMMING (A PRINCIPAL CONCERN IN THE DESIGN OF THE ADA LANGUAGE).

INSTRUCTOR NOTES

MENTION THAT STUDENTS WILL BE ASKED TO CRITIQUE THE INSTRUCTION AT THE END. ASK THEM TO JOT DOWN NOTES ON ANYTHING THAT WAS EXCEPTIONALLY CLEAR/UNCLEAR, AS WELL AS ANY IMPRESSIONS OR SUGGESTIONS THAT MIGHT IMPROVE THE MODULE.

POINT OUT THAT WE HAVE JUST COMPLETED THE PREFACE, AND ARE ABOUT TO START CONCEPTS AND ISSUES.

OUTLINE OF MODULE L103

SECTION	TITLE
1	PREFACE
2	CONCEPTS AND ISSUES
3	TYPES
4	STATEMENTS AND CONTROL STRUCTURES
5	PROGRAM UNITS
6	SEPARATE COMPILATION
7	SUMMARY

INSTRUCTOR NOTES

ALLOW 100 MINUTES TO COVER CONCEPTS AND ISSUES. THE FIRST BREAK SHOULD FOLLOW THE IMPLEMENTATION SECTION.

SECTION 2
CONCEPTS AND ISSUES

INSTRUCTOR NOTES

THESE ARE THE FIVE SUB-SECTIONS WE WILL COVER IN THIS SECTION.

DON'T SPEND MORE THAN 15 MINUTES DISCUSSING THE PROGRAM EXAMPLE.

CONCEPTS AND ISSUES

I. A TYPICAL PROGRAM WRITTEN IN A HIGH-ORDER LANGUAGE.

II. WHAT IS A HIGH-ORDER LANGUAGE?

III. CAPABILITIES OF HIGH-ORDER LANGUAGES.

IV. THE IMPLEMENTATION.

V. THE VIEWPOINT OF THE HIGH-ORDER LANGUAGE PROGRAMMER.

INSTRUCTOR NOTES

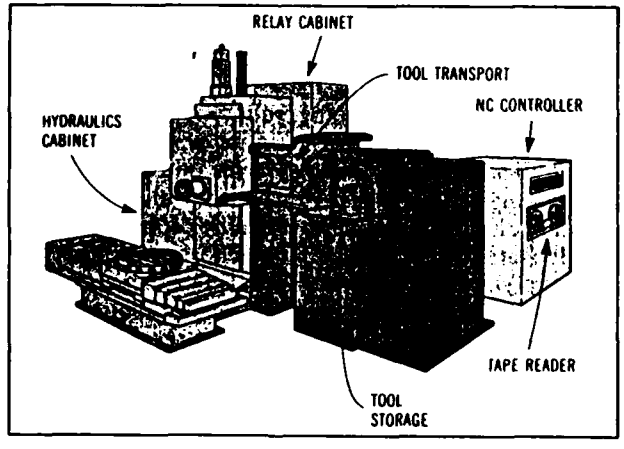
POINT OUT TO THE STUDENTS THAT THEY ARE NOT MEANT TO COMPLETELY UNDERSTAND THIS SECTION. IT IS PRESENTED HERE AS A BRIEF INTRODUCTION. ITS MAJOR POINT IS THAT EVEN THOUGH THEY DON'T KNOW THE HIGH-ORDER LANGUAGE OR PROCESS CONTROL SYSTEMS, THIS PROGRAM IS READABLE AND WHAT IT DOES IS FAIRLY CLEAR.

INSTRUCTOR SHOULD REVIEW THIS EXAMPLE AND THE DECLARATIONS FOR IT PRIOR TO CLASS.

I. A TYPICAL PROGRAM IN A HIGH-ORDER LANGUAGE SCENARIO

THIS CODE IS PART OF A LARGE SYSTEM CONTROLLING THE OPERATIONS OF AN AUTOMATED FACTORY. IT CONCERNS A SINGLE MACHINING WORKSTATION. PARTS ARE AUTOMATICALLY TRANSPORTED TO THE STATION, QUEUED, PROCESSED ONE-BY-ONE, AND PASSED BACK TO THE TRANSPORT SYSTEM.

THE MACHINE HOLDS A VARIETY OF TOOLS AND CAN TYPICALLY PERFORM SEVERAL TYPES OF OPERATIONS (E.G. DRILLING, MILLING, TAPPING). THE PART REQUIRES A SERIES OF OPERATIONS TO BE PERFORMED, AND STAYS AT THIS WORKSTATION UNTIL IT REACHES A STEP THAT THIS MACHINE CAN'T DO. THEN IT IS UNLOADED AND PASSED TO THE NEXT MACHINE.



INSTRUCTOR NOTES

FIRST, POINT OUT:

1. TEXTUAL STRUCTURE, IN PARTICULAR:
   (a) INDENTATION
   (b) END-OF-LINE TERMINATES COMMENTS ONLY (ALL OTHERS USE SEMICOLON)
   (c) "BRACKET" STRUCTURE: if - end if; begin -- end;
2. COMMENTS
3. ABSENCE OF LABELS, ADDRESSES, AND ANY REFERENCE TO MEMORY USAGE
4. STATEMENTS ARE MORE VARIED, COMPLEX, AND EXPRESSIVE THAN IN ASSEMBLER

DO A QUICK WALKTHROUGH. FOR EACH STATEMENT EXPLAIN WHAT IT DOES; DON'T SAY "THIS IS AN INDEXED COMPONENT," "THIS IS A PROCEDURE CALL." DEPENDING ON THE SOPHISTICATION OF THE CLASS, YOU MAY WISH, EITHER NOW OR AT THE END OF THE CLASS, TO HAND OUT COPIES OF THE DECLARATIONS NEEDED BY THIS PROGRAM (E.G., TO ILLUSTRATE THE STRUCTURE OF THE TYPE Part).

TYPICAL PROBLEM

```
with Machine_Shop_Pac;  use Machine_Shop_Pac;  -- contains definitions
procedure Process_Next_Part (Part_Queue  : Queue;
                             Workstation : NC_Machine) is
    This_Part          : Part;
    No_More_Parts      : Boolean;
    Current_Operation  : Operation_ID;
    Tool               : Rotary_Tool_Bit;
    Feed_Rate          : Inch_per_Min;          -- subtype of Float
    Cutting_Speed      : Surface_Ft_per_Min;    --    "    "    "
    Spindle_Speed      : RPM;                   -- subtype of Integer
begin -- Process_Next_Part
    Get_Part (Part_Queue, This_Part, No_More_Parts);
    if No_More_Parts then
        return;
    end if;
    Current_Operation := This_Part.Step (This_Part.Current);
    Load (This_Part, Workstation);
    while Compatible (Current_Operation, Workstation)
    loop
        Tool := Best_Tool (Current_Operation, Workstation);
        Optimize (Cutting_Speed, Feed_Rate,           -- find these
                  Tool.Material, This_Part.Material); -- given these
        Spindle_Speed := RPM(Cutting_Speed /
                             (Pi * Float (Tool.Diameter) / 12.0));
        Set_Up (Workstation, Tool, Spindle_Speed, Feed_Rate);
        Perform_Machining_Step (Current_Operation, Workstation);
    exit when This_Part.Current >= This_Part.Total_Steps;
        This_Part.Current := This_Part.Current + 1;
        Current_Operation := This_Part.Step (This_Part.Current);
    end loop;
    Unload (This_Part, Workstation);
end Process_Next_Part;
```

THE THIRD BULLET IS A LITTLE PROPAGANDA, BUT ALSO POINTS OUT A NECESSARY CHANGE OF MENTAL HABIT (FOR MANY).

IN ASSEMBLY LANGUAGE THESE BENEFITS CAN BE APPROXIMATED WITH WELL-DESIGNED MACROS.

MAIN POINTS

- HOL DOESN'T QUITE READ LIKE ENGLISH AT FIRST READING, BUT IT'S NOT TOO FAR FROM ENGLISH EITHER.

- ONE PAGE OF HOL SPECIFIES A LOT OF PROCESSING.

- HOL PROGRAM REFLECTS THE USER'S VIEWPOINT, NOT THE MACHINE'S.

INSTRUCTOR NOTES

ALLOW 15 MINUTES FOR LECTURE AND DISCUSSION IN THIS AREA.

CONCEPTS AND ISSUES

I. A TYPICAL PROGRAM WRITTEN IN A HIGH-ORDER LANGUAGE.

II. WHAT IS A HIGH-ORDER LANGUAGE?

III. CAPABILITIES OF HIGH-ORDER LANGUAGES.

IV. THE IMPLEMENTATION.

V. THE VIEWPOINT OF THE HIGH-ORDER LANGUAGE PROGRAMMER.

- MACHINE-INDEPENDENCE:

  a. PROGRAMMER DOES NOT NEED TO KNOW MACHINE LANGUAGE
  b. SAME LANGUAGE CAN BE IMPLEMENTED ON NUMEROUS DIFFERENT SYSTEMS
  c. THEORETICALLY, A PROGRAM WRITTEN IN A HIGH-ORDER LANGUAGE WILL RUN ON ANY MACHINE WITHOUT MODIFICATION (PORTABILITY). THIS HAS NOT BEEN ENTIRELY TRUE IN PRACTICE, EVEN WITH WIDELY-USED LANGUAGES

- DON'T BOG DOWN IN DISCUSSIONS ON PORTABILITY. IT WILL COME UP IN GREAT DETAIL LATER.

- DATA TYPES:

  THIS IS NOT TRUE FOR LANGUAGES LIKE LISP, BUT THOSE ARE OUTSIDE THE SCOPE OF THIS COURSE.

  THE PHYSICAL REPRESENTATION OF DATA CAN ALSO BE SPECIFIED, IF NECESSARY. SINCE THAT DECREASES PORTABILITY, IT SHOULD BE DONE WITH GREAT CARE.

- ANALYSIS BY COMPILER:

  FOR THE MOMENT, DEFINE COMPILER AS THE PROGRAM THAT TRANSLATES HIGH-ORDER LANGUAGE CODE INTO MACHINE LANGUAGE.

  INSTRUCTOR NOTA BENE: THE IDEA OF THE COMPILER ANALYZING THE SOFTWARE IS DISTASTEFUL TO MANY EXPERIENCED ASSEMBLY-LANGUAGE PROGRAMMERS, WHO DON'T WANT TO BE TOLD, ESPECIALLY BY THE COMPILER, WHAT THEY MAY OR MAY NOT DO.

WHAT IS A HIGH-ORDER LANGUAGE?

DEFINED BY ITS CHARACTERISTICS:

- INDEPENDENT OF MACHINE AND MACHINE LANGUAGE
- HIGH-ORDER INSTRUCTION SET (ONE STATEMENT IS EQUIVALENT TO MANY LINES OF MACHINE CODE).
- APPLICATION ORIENTED, RATHER THAN IMPLEMENTATION-ORIENTED
- DATA EXIST IN DIFFERENT <u>TYPES</u>

FUNCTIONAL DISTINCTION:	REAL VS CHARACTER
RATHER THAN	
MEMORY ACCESS DISTINCTION:	BYTE VS WORD

  THE TYPE DEFINES HOW A DATA ITEM IS INTENDED TO BE USED, NOT ITS PHYSICAL CHARACTERISTICS.

- COMPILER PERFORMS ANALYSIS OF SOFTWARE PRIOR TO EXECUTION, SEVERELY CONSTRAINS IMPLEMENTATION-DEPENDENT PRACTICES.

EXPLAIN THE EXAMPLE ASSIGNMENT STATEMENT.

GENERATIONS OF BEGINNER PROGRAMMERS HAVE BEEN SHOCKED BY

```
X = X + 1
```

THEREFORE, ADA AND OTHER LANGUAGES USE :=

THIS QUOTE FROM THE ORIGINAL REPORT EXPLAINS THE MOTIVATION BEHIND FORTRAN.

EVOLUTION OF HIGH-ORDER LANGUAGES

1. FORTRAN -- FIRST MAJOR HOL EFFORT, EMPHASIZED EXPRESSION ABSTRACTION:

$$A = B + C(I) + D/E$$

"THE IBM MATHEMATICAL FORMULA TRANSLATING SYSTEM OR BRIEFLY, FORTRAN, WILL COMPRISE A LARGE SET OF PROGRAMS TO ENABLE THE IBM 704 TO ACCEPT A CONCISE FORMULATION OF A PROBLEM IN TERMS OF A MATHEMATICAL NOTATION AND TO PRODUCE AUTOMATICALLY A HIGH-SPEED 704 PROGRAM FOR THE SOLUTION OF THE PROBLEM."

-- PROGRAMMING RESEARCH GROUP, APPLIED SCIENCE DIVISION, IBM, "PRELIMINARY REPORT, SPECIFICATIONS FOR THE IBM MATHEMATICAL FORmula TRANslating SYSTEM FORTRAN," 10 NOVEMBER 1954.

INSTRUCTOR NOTES

NOTE: THE LEFT HAND EXAMPLE IS AN ALGOL-LIKE STATEMENT. THE RIGHT HAND EXAMPLE IS A FORTRAN-LIKE SET OF STATEMENTS

(FOR THE INSTRUCTOR'S NOTE: THE LACK OF SEMICOLON IS NOT AN ERROR -- ALGOL DOES NOT REQUIRE IT.)

EVOLUTION OF HIGH-ORDER LANGUAGES

2. ALGOL -- EXHIBITS CONTROL ABSTRACTION, "STRUCTURE:"

```
if X = Y
then
   A := B
else
   A := C
end if
```

instead of:

```
      if X .EQ. Y go to 132
      A = C
      go to 150
132   A = B
150   continue
```

INSTRUCTOR NOTES

EVOLUTION OF HIGH-ORDER LANGUAGES

3. COBOL -- BUSINESS-ORIENTED, MEANT TO BE "ENGLISH-LIKE." ENCOURAGED VERBOSITY, EMPHASIZING READABILITY RATHER THAN WRITABILITY, A VIEW WHICH HAS RENEWED PROMINENCE TODAY. DEMANDED EXPLICIT DECLARATION OF ALL DATA, IN A SEPARATE PART OF THE PROGRAM TEXT.

4. JOVIAL, PL/I -- TRIED TO PUT TOGETHER FEATURES OF OTHER LANGUAGES TO MAKE A UNIVERSAL LANGUAGE FOR SCIENTIFIC, BUSINESS, AND SYSTEMS PROGRAMMING APPLICATIONS. JOVIAL WAS THE FIRST LANGUAGE USED FOR ITS OWN COMPILER.

INSTRUCTOR NOTES

"IN SUMMARY, HOLs EVOLVED TO MAKE LARGE SYSTEMS MORE MANAGEABLE. AS SYSTEMS GREW LARGER, THEY BECAME HARDER TO CONTROL AND MAINTAIN."

EVOLUTION OF HIGH-ORDER LANGUAGES

5. PASCAL, ADA -- EMPHASIZE DATA ABSTRACTION: OPERATIONS ON DATA ARE DEFINED WITHOUT REFERENCE TO A MAPPING OR OTHER IMPLEMENTATION, E.G., NON-NUMERIC TYPES EXIST EXPLICITLY WITHOUT BEING MAPPED TO NUMERIC TYPES (AT PROGRAMMER'S LEVEL). ADA TIES TOGETHER MANY KINDS OF DATA ABSTRACTION.

INSTRUCTOR NOTES

DESIDERATA - "THINGS THAT ARE DESIRABLE."

DESIDERATA FOR STATE-OF-THE-ART HIGH-ORDER LANGUAGES

1. READABILITY, CLARITY, SELF-DOCUMENTING STRUCTURE (NO SUBSTITUTE FOR READABLE STYLE, HOWEVER)

2. MAINTAINABILITY

3. SUPPORT FOR STRUCTURED PROGRAMMING

4. SUPPORT FOR NUMEROUS DATA TYPES AND STRUCTURES: STRONG TYPING

5. MODULARITY OF PROGRAMS

INSTRUCTOR NOTES

ALLOW 20 MINUTES FOR LECTURE AND DISCUSSION IN THIS SECTION.

CONCEPTS AND ISSUES

I. A TYPICAL PROGRAM WRITTEN IN A HIGH-ORDER LANGUAGE.

II. WHAT IS A HIGH-ORDER LANGUAGE?

III. CAPABILITIES OF HIGH-ORDER LANGUAGES.

IV. THE IMPLEMENTATION.

V. THE VIEWPOINT OF THE HIGH-ORDER LANGUAGE PROGRAMMER.

INSTRUCTOR NOTES

IN A LITERAL SENSE, ANYTHING THAT CAN BE DONE IN A HIGH-ORDER LANGUAGE COULD BE DONE IN ASSEMBLY LANGUAGE. HOWEVER, PROGRAM AND DATA STRUCTURES ARE SEEN FROM QUITE A DIFFERENT VIEWPOINT (CLOSER TO THE APPLICATION ANALYSIS), WHICH IMPELS A DIFFERENT DESIGN APPROACH.

"CAPABILITIES" HERE MEANS MORE THAN JUST THE OPERATIONS AVAILABLE FOR MANIPULATING DATA.

BEWARE OF THE TERM "READABLE"! TO MANY, THEY ARE NOT "READABLE" AT ALL. BY READABLE, WE MEAN THAT THE FUNCTION OF THE PROGRAM IS EASILY PERCEIVED BY THE USER.

CAPABILITIES OF HIGH-ORDER LANGUAGES

- READABILITY
- PORTABILITY
- REUSABILITY
- MAINTAINABILITY

INSTRUCTOR NOTES

READABILITY

THE PROGRAMMER IS COMMUNICATING NOT ONLY WITH THE MACHINE BUT WITH HUMAN BEINGS WHO NEED TO UNDERSTAND THE CODE. THEY INCLUDE:

- COLLEAGUES WORKING ON THE SAME PROJECT
- PERSONS ADAPTING THE CODE FOR DIFFERENT USES
- PERSONS WRITING OTHER CODE WHICH NEEDS TO INTERFACE WITH THIS CODE
- PERSONS MAINTAINING THE CODE WHEN THE PROGRAMMER HAS MOVED ON TO OTHER THINGS
- THE PROGRAMMER HIMSELF, DEBUGGING THE CODE, OR SIMPLY RETURNING TO WORK ON IT AFTER DOING SOMETHING ELSE

INSTRUCTOR NOTES

"THE ATTENTION NEEDED TO MAKE THE PROGRAM READABLE (I.E. PICKING MNEMONIC NAMES, INDENTING STATEMENTS, COMMENTING) NECESSARILY MAKES IT "HARDER TO WRITE."

READABILITY

HIGH-ORDER LANGUAGES ARE INHERENTLY MORE READABLE TO HUMANS. THEY ARE NOT, HOWEVER, NECESSARILY MORE WRITABLE; THIS DEPENDS ON THE LEVEL OF ABSTRACTION OF THE APPLICATION.

INSTRUCTOR NOTES

READABILITY

NO LANGUAGE CAN BY ITSELF MAKE SOFTWARE READABLE: A CONSISTENT SET OF STYLE AND DOCUMENTATION CONVENTIONS IS STILL NECESSARY. HOWEVER, BY HIDING THE IMPLEMENTATION CONSIDERATIONS, THE LANGUAGE REMOVES A PRIMARY IMPEDIMENT TO CLARITY.

OF COURSE THE PROGRAM IS CHANGED TO RUN ON THE DIFFERENT MACHINE -- THE POINT IS THAT THESE CHANGES ARE NONE OF THE PROGRAMMER'S CONCERN. IN FACT, THEY ARE INVISIBLE AT THE HIGH-ORDER LANGUAGE LEVEL.

PORTABILITY

"CAN I RECOMPILE THIS PROGRAM AND RUN IT ON A DIFFERENT MACHINE WITHOUT NEEDING TO CHANGE IT?"

- AVOIDANCE - PREVIOUS NOTE MENTIONED CONSTRAINTS IMPOSED BY COMPILER E.G., CAN'T MAKE USE OF KNOWLEDGE OF HOW CHARACTERS ARE STORED TO PERFORM ARITHMETIC OPERATIONS ON THEM. COMPILER CAN'T POLICE EVERYTHING: LANGUAGE DEFINES CERTAIN PRACTICES AS ERRONEOUS: THEY ARE NOT GUARANTEED TO WORK WHEN HARDWARE OR OPERATING SYSTEM CHANGES.

- ENFORCEMENT - WITH PREVIOUS LANGUAGES, PORTABILITY WAS NEVER REALLY ACHIEVED: EVERY COMPILER MANUFACTURER HAD OWN "DIALECT." DoD HOPES FOR BETTER ENFORCEMENT WITH ADA - THE NAME ADA IS TRADEMARKED, AND ONLY COMPILERS THAT PASS A RIGOROUS SERIES OF TESTS CAN BE CALLED ADA COMPILERS.

PORTABILITY DEPENDS ON

- SEPARATION OF LANGUAGE DEFINITION AND IMPLEMENTATIONS (LANGUAGE DESIGNER'S RESPONSIBILITY)

- AVOIDANCE OF IMPLEMENTATION-DEPENDENT CODE (PROGRAMMER'S RESPONSIBILITY)

- STRICT ENFORCEMENT OF STANDARDS -- COMPILER MUST SUPPORT LANGUAGE AS DEFINED (COMMUNITY RESPONSIBILITY)

INSTRUCTOR NOTES

REUSABILITY

"MUCH EFFORT WENT INTO THE DEVELOPMENT OF THE NAVIGATION PACKAGE FOR THIS AIRCRAFT. CAN AT LEAST PART OF IT BE REUSED FOR THE NEXT MODEL?"

YES, IF:

- THE SOFTWARE IS PORTABLE
- THE PROGRAM DESIGN IS MODULAR
- EACH MODULE WAS DEFINED AND DEVELOPED WITH REUSABILITY IN MIND: SPECIFIC FUNCTION, GENERAL APPLICATION

INSTRUCTOR NOTES

WE WILL SEE THIS AGAIN LATER IN THE ADA OVERVIEW WHEN WE TALK ABOUT PACKAGES.

REUSABILITY

MODULARITY IS BUILT IN TO THE MORE MODERN LANGUAGES, IMPELLING THE SOFTWARE DESIGNER TO THINK IN TERMS OF "BUILDING BLOCKS" WITH CAREFULLY SPECIFIED INTERFACES.

POINT OUT HERE THAT ASSEMBLY LANGUAGE, FOR EXAMPLE, HAS ONE SUCH FEATURE, THE LABEL. IT HAS NO FUNCTION DURING PROGRAM EXECUTION. IT IS A DEVELOPMENT AND MAINTENANCE FEATURE. A MACRO IS ANOTHER SUCH FEATURE. THE HOL HAS MANY OF THESE, WHICH WE WILL SEE LATER.

MAINTAINABILITY

HOL'S OFFER MANY FEATURES THAT DO NOT AFFECT PROGRAM EXECUTION, BUT AID DEVELOPMENT AND MAINTENANCE.

WHY IS A HIGH-ORDER LANGUAGE HARDER TO LEARN?

- CONTAINS MANY ABSTRACT CONCEPTS WHICH DON'T EXIST IN ASSEMBLY LANGUAGE, E.G. DATA TYPES, DYNAMIC STORAGE ALLOCATION, FUNCTION SUBPROGRAMS,
- THIS IS NOTHING NEW; TOOLS THAT DO MORE ARE USUALLY HARDER TO LEARN.

SO WHY BOTHER?

- THE INITIAL LEARNING INVESTMENT PAYS OFF LATER, WHEN THE TOOL IS PUT TO USE.

DRAWBACKS OF HIGH-ORDER LANGUAGES

WHILE THE BASIC SYNTAX IS EASY TO LEARN (MADE TO RESEMBLE ENGLISH AND ALGEBRA), A HIGH-ORDER LANGUAGE IS AS A WHOLE MORE COMPLEX AND HARDER TO LEARN THAN AN ASSEMBLY LANGUAGE.

Q. WHAT DO YOU MEAN, "THIS IS DELIBERATE;" WHY?

A. USING "SPECIAL FEATURES" OF THE HARDWARE BY DEFINITION MAKES YOUR PROGRAM IMPLEMENTATION-DEPENDENT: IT IS TIED NOT ONLY TO THAT HARDWARE, BUT POSSIBLY TO A SPECIFIC VERSION OF THE HARDWARE. IT IS NO LONGER PORTABLE, AND PROBABLY NOT REUSABLE. ALSO, IT IS NOT UNDERSTANDABLE BY SOMEBODY NOT FAMILIAR WITH THE IDIOSYNCRASIES OF THE HARDWARE.

FOR SITUATIONS WHERE THERE IS NO GENERAL, PORTABLE SOLUTION, OR WHERE THE ADVANTAGES OF THE SPECIAL FEATURES OUTWEIGH OTHER CONSIDERATIONS, ADA PROVIDES A MECHANISM FOR "LOW-LEVEL CODE" WHICH ATTEMPTS TO CONTAIN THE HARDWARE-DEPENDENT ASPECTS AS MUCH AS POSSIBLE, SO THE REST OF THE CODE STAYS GENERAL.

THE HIGH-ORDER LANGUAGE

MAY NOT PROVIDE ACCESS, AS EASILY AS ASSEMBLY LANGUAGE, TO SPECIAL FEATURES OF THE HARDWARE. (THIS IS DELIBERATE).

INSTRUCTOR NOTES

Q. HMMPH! WHAT OTHER BENEFITS?

A. FOR EXAMPLE: SOFTWARE DEVELOPMENT TIME, FROM REQUIREMENTS SPECIFICATION TO OPERATIONAL, DEBUGGED*, DOCUMENTED PRODUCT, IS SHORTER. FOR LARGE PROGRAMMING SYSTEMS PRODUCTS INVOLVING MANY INDIVIDUALS AND MANY SUBSYSTEMS, THE DIFFERENCE IS GREAT.

THE BOTTOM LINE IS THAT OBJECT-CODE EFFICIENCY IS SIMPLY NO LONGER THE ONLY IMPORTANT ISSUE -- BETTER ALGORITHM DESIGN MAKES MUCH MORE OF A DIFFERENCE THAN COMPILATION VS. ASSEMBLY.

N.B.: THE FIGURES ARE ONLY INDICATIVE. FOR CERTAIN EXOTIC LANGUAGES, THE LOSS OF EFFICIENCY CAN BE SEVERE.

*ASSUME BOTH PRODUCTS ARE DEBUGGED TO THE SAME DEGREE.

THE QUESTION WE'VE ALL BEEN WAITING FOR

MUCH CAN BE SAID ABOUT OBJECT CODE EFFICIENCY OF HIGH-ORDER VS. ASSEMBLY LANGUAGE; SUFFICE IT TO SAY THAT:

(a) THEY ARE ROUGHLY COMPARABLE (I.E., THE HIGH-ORDER LANGUAGE WILL NOT GENERATE CODE THAT IS TEN TIMES SLOWER OR BIGGER: TYPICAL LOSS OF EFFICIENCY IS MORE OF THE ORDER OF 5-10%)

(b) CONSCIOUS DECISION: INEFFICIENCIES THAT DO EXIST ARE DEEMED ACCEPTABLE IN VIEW OF OTHER BENEFITS

INSTRUCTOR NOTES

ALLOW 20 MINUTES FOR LECTURE AND DISCUSSION IN THIS SECTION.

TAKE A SHORT BREAK AFTER THIS SECTION.

CONCEPTS AND ISSUES

I. A TYPICAL PROGRAM WRITTEN IN A HIGH-ORDER LANGUAGE.

II. WHAT IS A HIGH-ORDER LANGUAGE?

III. CAPABILITIES OF HIGH-ORDER LANGUAGES.

IV. THE IMPLEMENTATION.

V. THE VIEWPOINT OF THE HIGH-ORDER LANGUAGE PROGRAMMER.

INSTRUCTOR NOTES

THE IMPLEMENTATION

THE COMPILER IS A PROGRAM WHICH TRANSLATES A SOURCE PROGRAM WRITTEN IN A HIGH-ORDER LANGUAGE TO A FUNCTIONALLY EQUIVALENT OBJECT PROGRAM IN MACHINE CODE.

INSTRUCTOR NOTES

PHASES OF COMPILATION

1. LEXICAL ANALYSIS

   - PARSE SOURCE PROGRAM INTO TOKENS (BASIC ELEMENTS OF LANGUAGE)
   - BUILD TABLES OF IDENTIFIERS

INSTRUCTOR NOTES

PHASES OF COMPILATION

2. SYNTACTIC ANALYSIS/SEMANTIC ANALYSIS

- RECOGNIZE SYNTACTIC CONSTRUCTS (STATEMENTS)
- INTERPRET MEANING, GENERATE INTERMEDIATE LANGUAGE EQUIVALENT

THE EXAMPLE IS EXAGGERATED IN THE SENSE THAT NOBODY WRITES CODE THAT BADLY. THE IMPORTANT POINT IS THAT A LOT LESS TIME NEEDS TO BE SPENT IN MINIMIZING THE COMPUTATIONAL STEPS: LET THE COMPILER DO IT.

BY THE WAY, EVEN IN ASSEMBLY LANGUAGE REDUNDANCIES MAY BE INTRODUCED BY MACROS (E.G., TWO MACRO CALLS END UP COMPUTING A CERTAIN ADDRESS TWICE).

OPTIMIZATIONS CAN BE VERY SOPHISTICATED.

PHASES OF COMPILATION

3. MACHINE-INDEPENDENT OPTIMIZATION

- IMPROVE INTERMEDIATE LANGUAGE PROGRAM TO REMOVE REDUNDANCIES, E.G.,

```
A := B + C          A := B + C
D := B + C    ---->  D := A
```

INSTRUCTOR NOTES

PHASES OF COMPILATION

4. STORAGE ASSIGNMENT

- DEFINE STORAGE REQUIREMENTS FOR DATA, INCLUDING INTERMEDIATE RESULTS

- STORAGE IS NOT NECESSARILY ALLOCATED AT COMPILE TIME, BUT REQUIREMENTS NEED TO BE KNOWN WHEN IT IS ALLOCATED.

THE ASSEMBLY PROCESS TAKES THE ASSEMBLY-LANGUAGE CODE AND TRANSLATES IT INTO MACHINE CODE.

IN THINGS LIKE REGISTER ALLOCATION, A COMPILER IS OFTEN MORE CLEVER THAN A MEDIUM-CALIBER PROGRAMMER. A COMPILER CANNOT OUTSMART A BRILLIANT ASSEMBLY PROGRAMMER, BUT IT IS REASONABLY GOOD, AND MUCH, MUCH FASTER.

PHASES OF COMPILATION

5. CODE GENERATION AND MACHINE-DEPENDENT OPTIMIZATION

- CONVERT INTERMEDIATE LANGUAGE TO MACHINE LANGUAGE OF TARGET SYSTEM
- OPTIMIZE TO MAKE EFFICIENT USE OF REGISTERS, ETC.
- TYPICALLY PRODUCES ASSEMBLY-LANGUAGE CODE (EASIER TO RESOLVE LABEL REFERENCES)

6. ASSEMBLY

INSTRUCTOR NOTES

LOAD MODULES WILL BE DISCUSSED SHORTLY.

INTERPRETERS

THERE EXIST PROGRAMS WHICH ANALYZE THE SOURCE CODE AND EXECUTE IT DIRECTLY, WITHOUT GENERATING AN OBJECT PROGRAM. MOST FAMILIAR IS IMPLEMENTATION OF BASIC LANGUAGE IN PERSONAL COMPUTERS.

INTERPRETERS ARE CONVENIENT (THEY ALLOW INTERACTIVE EDITING AND DISPENSE WITH THE NEED TO MAINTAIN OBJECT PROGRAMS AND LOAD MODULES) BUT THEY ARE SLOW.

INSTRUCTOR NOTES

THIS IS A GOOD PLACE TO BREAK.

LINKING

- ESSENTIALLY SAME PROCESS AS WITH ASSEMBLY LANGUAGE
- BINDS SEPARATELY-COMPILED MODULES, RESOLVES REFERENCES
- SINCE TARGET MACHINE IS NOT NECESSARILY SAME AS HOST, A FURTHER STEP, CALLED EXPORTING, MAY BE REQUIRED TO GENERATE AN EXECUTABLE LOAD MODULE

INSTRUCTOR NOTES

ALLOW 30 MINUTES FOR THIS SECTION.

CONCEPTS AND ISSUES

I. A TYPICAL PROGRAM WRITTEN IN A HIGH-ORDER LANGUAGE.

II. WHAT IS A HIGH-ORDER LANGUAGE?

III. CAPABILITIES OF HIGH-ORDER LANGUAGES.

IV. THE IMPLEMENTATION.

V. THE VIEWPOINT OF THE HIGH-ORDER LANGUAGE PROGRAMMER.

INSTRUCTOR NOTES

"THE ADA MACHINE"

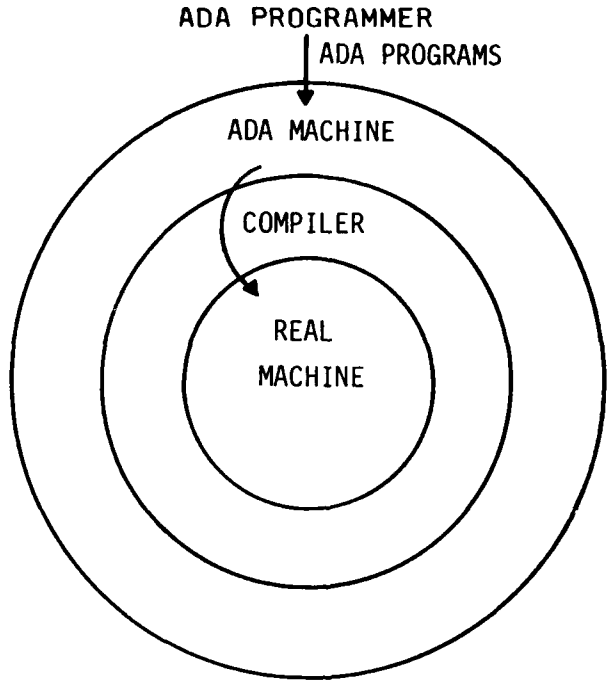
- COMPILER AND VARIOUS PROGRAM DEVELOPMENT TOOLS CONCEPTUALLY FORM A SHELL SURROUNDING THE REAL MACHINE.

- AS FAR AS THE PROGRAMMER CAN TELL, THE HIGH-ORDER LANGUAGE IS THE ONLY LANGUAGE.

- THE PROGRAMMER DOES NOT SEE THE REAL MACHINE OR ITS LANGUAGE -- IN FACT, UNTIL THE PROGRAM IS COMPILED THERE NEED BE NO REAL MACHINE. THE SAME PROGRAM CAN BE RUN ON DIFFERENT MACHINES WITH NO CHANGE APPARENT TO THE PROGRAMMER.

THE VIEWPOINT OF THE HIGH-ORDER LANGUAGE PROGRAMMER

"THE ADA MACHINE"



INSTRUCTOR NOTES

THESE POWERFUL OP CODES ARE THE "CONTROL STRUCTURES" WHICH WE WILL SEE LATER. ALL COMPUTERS HAVE CONTROL STATEMENTS. THE ADA MACHINE HAS ARBITRARILY COMPLEX ONES.

THE LANGUAGE IS THAT OF THE APPLICATION

- DATA AND DECISION STRUCTURES REFLECT PROBLEM SOLUTION CONSIDERATIONS. THE ALGORITHM IS MORE EXPLICIT, SINCE MANY IMPLEMENTATION ISSUES ARE HIDDEN.

- EVERYTHING IS CONCEPTUALLY MEMORY TO MEMORY

- MEMORY MANAGEMENT IS AUTOMATIC. CONCEPTUALLY, THE ADA MACHINE HAS INFINITE MEMORY.

- THE ADA MACHINE HAS MANY POWERFUL OP CODES.

INSTRUCTOR NOTES

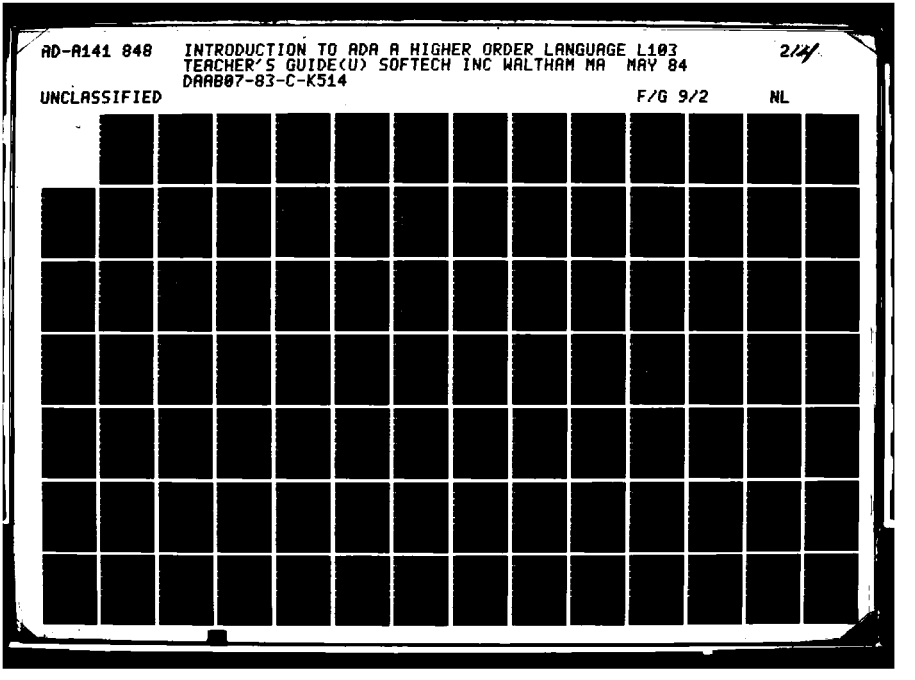
WHY IS IT SO IMPORTANT THAT THE PROGRAM REFLECT THE USER'S VIEWPOINT? BECAUSE IT'S BETTER (AND HARDER) TO BE CORRECT THAN TO BE EFFICIENT.

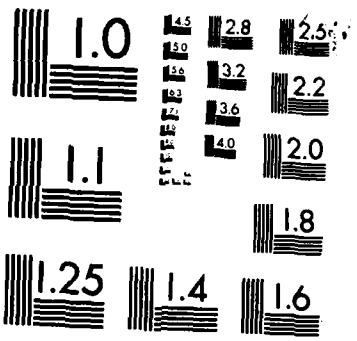
READABILITY IS A CONSTANT CONCERN

- CLEAR STRUCTURE
- APPLICATION-ORIENTED NAMING CONVENTIONS
- CLEAR DOCUMENTATION
- CONSISTENT, "DE FACTO STANDARD" STYLE CONVENTIONS

INSTRUCTOR NOTES

THIS MAY BE A HARD THING FOR ASSEMBLY PROGRAMMERS TO GET USED TO.


AD-A141 848 INTRODUCTION TO ADA A HIGHER ORDER LANGUAGE L103 TEACHER'S GUIDE(U) SOFTECH INC WALTHAM MA MAY 84 DAAB07-83-C-K514
2/4
UNCLASSIFIED
F/G 9/2
NL



REUSABILITY

PROGRAMMER IS NOT ISOLATED -- WE ALL USE EACH OTHER'S PRODUCTS

- MODULARITY
- GENERALITY
- INTERFACE STANDARDIZATION
- WITH A TRULY STANDARDIZED LANGUAGE THE LONG HOPED-FOR IDEAL OF "OFF-THE-SHELF" PURCHASED SOFTWARE COMPONENTS CAN BE REALIZED.

INSTRUCTOR NOTES

ALLOW 60 MINUTES FOR THIS SECTION.

BREAK FOR LUNCH AFTER THIS SECTION.

SECTION 3
DATA TYPES

INSTRUCTOR NOTES

NOW WE START TALKING ABOUT THE "ADA MACHINE."

TYPES

THE MEMORY ELEMENTS OF THE ADA MACHINE ARE CALLED OBJECTS.

IN ADDITION TO A VALUE, EVERY OBJECT HAS A PROPERTY CALLED ITS TYPE, WHICH CONSTRAINS THE WAY IN WHICH THE OBJECT MAY BE USED.

THIS IS NOT ENTIRELY NEW. HARDWARE HAS TYPES (BYTE, WORD, REGISTER). THE ADA MACHINE HAS MANY MORE.

A TYPE IS CHARACTERIZED BY:

(0 - 15)

- A SET OF PERMISSIBLE VALUES (E.G. A 4-BIT BYTE CAN HAVE THE VALUES 0000-1111)
- A SET OF OPERATIONS APPLICABLE TO THOSE VALUES (E.G. THERE IS NO OP CODE TO USE A BYTE AS A POINTER)

THE ADA MACHINE'S TYPES HAVE SIMILAR PROPERTIES.

INSTRUCTOR NOTES

HARDWARE TYPES VS. ADA TYPES

A HYPOTHETICAL MACHINE:

OPERATION ⟶

HARDWARE TYPE ↓

	ADD	AND	FLOAT ADD	USE AS ADDRESS
BYTE	-	✓	-	-
WORD	✓	✓	-	-
REGISTER	✓	✓	✓	✓
FLOAT REGISTER	-	-	✓	-

TYPICALLY, APPLICABILITY OF OPERATION IS RESTRICTED ONLY BY HARDWARE IMPLEMENTATION DIFFICULTY.

INSTRUCTOR NOTES

THE MAIN DIFFERENCE

HARDWARE: OPERATIONS CAN BE APPLIED IF OPERANDS HAVE THE PROPER PHYSICAL CHARACTERISTICS.

EXAMPLES: FLOATING POINT ADDITION REQUIRES A 32-BIT OPERAND.

THE RESULT MAY LATER BE USED AS AN ADDRESS.

ADA: OPERATIONS CAN BE APPLIED ONLY IF OPERANDS HAVE THE PROPER TYPE.

EXAMPLE: AN OBJECT OF TYPE FLOAT CANNOT BE TREATED AS AN ADDRESS.

INSTRUCTOR NOTES

SOME PREDEFINED ADA TYPES

TYPE	VALUES	OPERATIONS
Integer	0 105 1_752 ...	+ - * / ...
Boolean	True False	not and or xor ...
Float	3.7 96.8 ...	+ - * / ...

THE PHYSICAL REPRESENTATION CAN BE SPECIFIED, IF ABSOLUTELY NECESSARY. ORDINARILY IT ISN'T.

ADA HAS A RICH SET OF BUILT-IN TYPES; IT ALSO ENCOURAGES YOU TO DEFINE YOUR OWN.

TYPE DECLARATIONS

THE TYPE OF A DATA ITEM DEFINES THE KIND OF INFORMATION CONTAINED (NOT THE AMOUNT OF MEMORY USED).

A MEMORY LOCATION MAY REPRESENT

- An Integer
- A Character
- A Day_Of_The_Week
- A Color

THE TYPE SPECIFIES WHICH

INSTRUCTOR NOTES

TYPE DECLARATIONS

A DECLARATION OF A TYPE IN THE ADA MACHINE CREATES A BLUEPRINT WHICH

- DEFINES THE SET OF VALUES A VARIABLE OF THAT TYPE CAN ASSUME (E.G., Integers, Booleans).

- DEFINES THE SET OF OPERATIONS WHICH CAN BE USED WITH VARIABLES OR CONSTANTS OF THAT TYPE (E.G., ADDITION, LOGICAL AND).

INSTRUCTOR NOTES

THESE ARE THE BLUEPRINTS FOR OBJECTS TO BE CREATED.

THE COMPILER FIGURES OUT:

- HOW MUCH MEMORY TO USE
- HOW TO REPRESENT THE VALUES AS BIT PATTERNS
- WHAT MACHINE INSTRUCTIONS TO USE FOR THE OPERATIONS

TYPE DECLARATION EXAMPLES

INTEGER TYPE

```
type Count is range 0 .. 1000;
type Coordinate is range -20 .. +20;
```

FIXED POINT TYPE

```
type Metric_Drill_Size is delta 0.1 range 1.0 .. 15.0;
type Money Is delta 0.01 range 0.01 .. 10_000.00;
```

FLOATING POINT TYPE

```
type Length is digits 6;
type Extra_Precise is digits 8;
```

ENUMERATION TYPE

```
type Switch_Position is (Hot, Off, Cold);
type Raw_Material is (Cast_Iron, Cold_Rolled_Steel, High_Speed_Steel,
                      Tungsten_Carbide, Stainless_Steel, Aluminum, Brass);
```

IN THE ADA MACHINE, DATA CAN BE ARBITRARILY BIG AND COMPLEX.

OBJECTS OF TYPE Part_Rec WILL CONSIST OF THREE COMPONENTS. TYPE DECLARATION DEFINES LOGICAL GROUPING. THE COMPILER FINDS A REPRESENTATION AND TAKES CARE OF THE BOOKKEEPING.

OBJECTS OF TYPE Data_Bank ARE TABLES OF CONSECUTIVE Part_Rec OBJECTS. INDIVIDUAL COMPONENTS OF THE ARRAY CAN BE ADDRESSED IN A RANDOM-ACCESS STYLE.

THE THIRD DECLARATION GIVES A MEANS TO CREATE AND DESTROY DYNAMICALLY Part_Rec OBJECTS. EXAMPLE WILL BE GIVEN LATER.

TYPE DECLARATION EXAMPLES (Continued)

RECORD TYPE

```
type Part_Rec is record
    Drill_Size : Metric_Drill_Size;
    Precision  : Extra_Precise;
    Material   : Raw_Material;
end record;
```

ARRAY TYPE

```
type Data_Bank is array (Count) of Part_Rec;
```

ACCESS TYPE

```
type Part_Pointer is access Part_Rec;
```

INSTRUCTOR NOTES

TYPES

WHEN WE HAVE A VARIABLE OF SOME PARTICULAR TYPE IT MUST HAVE THE CONSTRAINTS PRESCRIBED BY THE TYPE DEFINITION

IF A VARIABLE IS OF TYPE Count THE ONLY VALUES IT CAN ASSUME ARE INTEGER VALUES BETWEEN 0 AND 1000 INCLUSIVE

LIKEWISE

IF A VARIABLE IS OF TYPE Switch_Position THEN THE ONLY VALUES IT CAN ASSUME ARE Hot, Off, OR Cold

INSTRUCTOR NOTES

TYPES

SIMILARLY

THE OPERATIONS THAT CAN BE PERFORMED WITH A VARIABLE ARE DETERMINED BY THE TYPE

EXAMPLE:

ADDITION, SUBTRACTION, MULTIPLICATION, AND DIVISION ARE DEFINED FOR THE INTEGER TYPE Count.

ADDITION IS NOT DEFINED FOR THE ENUMERATION TYPE Raw_Material

THEY ARE THE MEMORY CELLS OF THE ADA MACHINE.

OBJECTS ARE VARIABLES AND CONSTANTS

INSTRUCTOR NOTES

IN THE ADA MACHINE, OBJECTS ARE CREATED AND DESTROYED AS NEEDED.

OBJECT DECLARATIONS

THE DECLARATION OF AN OBJECT:

- SETS ASIDE MEMORY FOR VALUES TO BE STORED IN
- ASSOCIATES THE VARIABLE OR CONSTANT WITH A SPECIFIC TYPE

EVERY OBJECT MUST BE ASSOCIATED WITH A SPECIFIC TYPE.

INSTRUCTOR NOTES

OBJECT DECLARATION EXAMPLES

```
X_Coordinate    : Coordinate;
Page_Count      : Count;
Hole_Diameter   : Metric_Drill_Size;
Account_Balance : Money;
Switch          : Switch_Position;
Machine_Part    : Part_Rec;
Inventory       : Data_Bank;
Current_Part    : Part_Pointer;
```

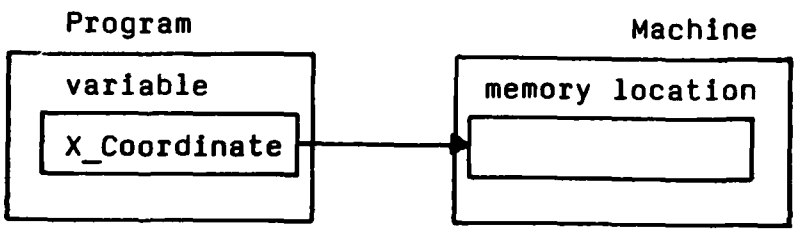
INSTRUCTOR NOTES

N.B.: SMALL EXAMPLES GIVE THE IMPRESSION THAT TYPE DECLARATIONS AND OBJECT DECLARATIONS GO IN PAIRS. IN LARGE PROGRAMS THERE WILL BE MANY OBJECTS OF ANY GIVEN TYPE.

OBJECT DECLARATIONS

```
type Coordinate is range -20 .. +20;
X_Coordinate : Coordinate;
```

THE OBJECT X_Coordinate IS A VARIABLE NAME ASSOCIATED WITH A MEMORY LOCATION THAT HAS BEEN SET ASIDE FOR AN INTEGER:



NO VALUE IS CURRENTLY ASSIGNED TO THIS MEMORY LOCATION.

THIS OBJECT IS OF THE TYPE Coordinate WHICH MEANS THAT THE VALUES WHICH THE VARIABLE X_Coordinate CAN ASSUME ARE -20 .. +20 INCLUSIVE.

THIS OBJECT CAN BE USED ONLY WITH THE SPECIFIC OPERATIONS DEFINED FOR INTEGER TYPES:

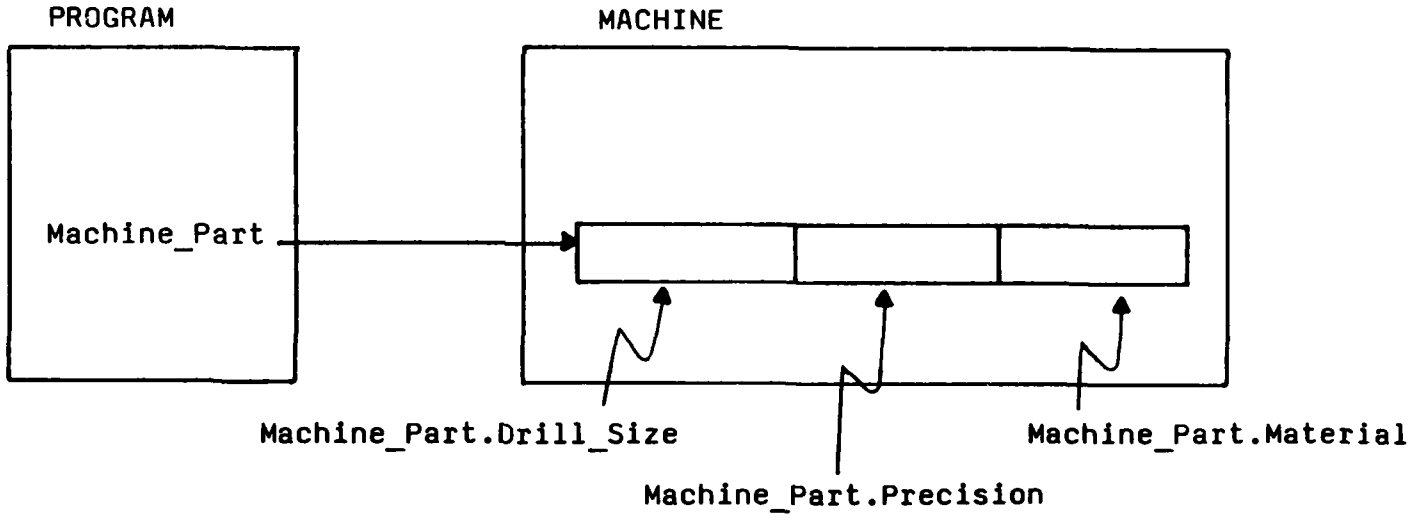
ARITHMETIC	+	-	*	/	**	
RELATIONAL	<	>	=	/=	>=	<=

INSTRUCTOR NOTES

OBJECT DECLARATIONS (Continued)

```
type Part_Rec is record
    Drill_Size : Metric_Drill_Size;
    Precision  : Extra_Precision;
    Material   : Raw_Material;
end record;
Machine_Part   : Part_Rec;
```

THIS OBJECT IS A VARIABLE ASSOCIATED WITH A MEMORY LOCATION THAT HAS BEEN SET ASIDE FOR A Part_Rec.



INSTRUCTOR NOTES

SHOW THE DECLARATION. DESCRIBE WHAT HAPPENS IN THE ADA MACHINE AS A RESULT OF ELABORATIONS.

TYPICAL PROBLEM

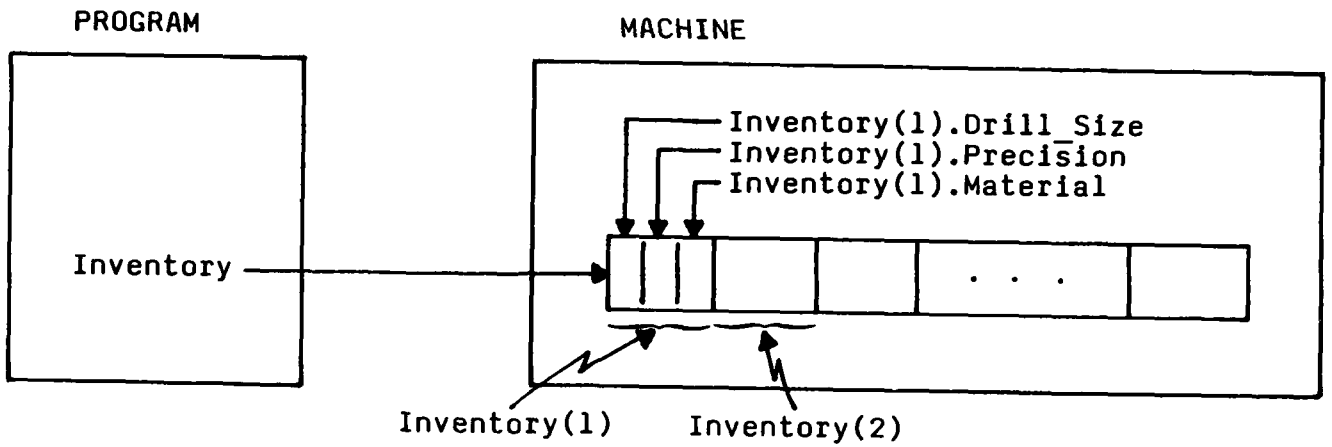
```
with Machine_Shop_Pac;  use Machine_Shop_Pac;  -- contains definitions
procedure Process_Next_Part (Part_Queue  : Queue;
                             Workstation : NC_Machine) is

    This_Part          : Part;
    No_More_Parts      : Boolean;
    Current_Operation  : Operation_ID;
    Tool               : Rotary_Tool_Bit;
    Feed_Rate          : Inch_per_Min;          -- subtype of Float
    Cutting_Speed      : Surface_Ft_per_Min;    --    "    "    "
    Spindle_Speed      : RPM;                   -- subtype of Integer

begin -- Process_Next_Part

    Get_Part (Part_Queue, This_Part, No_More_Parts);
    if No_More_Parts then
        return;
    end if;

    Current_Operation := This_Part.Step (This_Part.Current);
    Load (This_Part, Workstation);
    while Compatible (Current_Operation, Workstation)
    loop
        Tool := Best_Tool (Current_Operation, Workstation);
        Optimize (Cutting_Speed, Feed_Rate,           -- find these
                  Tool.Material, This_Part.Material); -- given these
        Spindle_Speed := RPM(Cutting_Speed /
                             (Pi * Float (Tool.Diameter) / 12.0));
        Set_Up (Workstation, Tool, Spindle_Speed, Feed_Rate);
        Perform_Machining_Step (Current_Operation, Workstation);

    exit when This_Part.Current >= This_Part.Total_Steps;

        This_Part.Current := This_Part.Current + 1;
        Current_Operation := This_Part.Step (This_Part.Current);
    end loop;
    Unload (This_Part, Workstation);
end Process_Next_Part;
```

INSTRUCTOR NOTES

OBJECT DECLARATIONS (Continued)

```
type Count is range 1 .. 1000;
type Data_Bank is array (Count) of Part_Rec;
Inventory : Data_Bank;
```

AN OBJECT OF type Data_Bank IS A SEQUENCE OF 1000 OBJECTS OF type Data_Bank.



N.B.: IN THE ADA MACHINE, THE COMPONENTS ARE ACCESSED USING INDEXED NOTATION, NOT POINTERS.

INSTRUCTOR NOTES

STRONG TYPING

DATA ITEMS OF DIFFERENT TYPES CAN NOT BE MIXED IN EXPRESSIONS NOR CAN VALUES OF ONE TYPE BE ASSIGNED TO VARIABLES OF A DIFFERENT TYPE.

WITHOUT STRONG TYPING:

```
Kilograms_1, Kilograms_2 : Integer;                -- KILOGRAMS
Pounds_1,    Pounds_2    : Integer;                -- POUNDS

Kilograms_1 :=  Kilograms_1 + Kilograms_2;         -- LEGAL AND LOGICAL
Pounds_1    :=  Pounds_1    + Pounds_2;            -- LEGAL AND LOGICAL
Pounds_1    :=  Kilograms_1 + Pounds_2;            -- LEGAL BUT NOT LOGICAL INTENT
```

INSTRUCTOR NOTES

DON'T BE SHOCKED BY THE ATTRIBUTES. WE ARE SUPPOSED TO DO SOME HANDWAVING IN THIS COURSE. DETAILS WILL BE COVERED IN L202.

STRONG TYPING (Continued)

WITH STRONG TYPING

```
type Kilogram is range 0 .. Integer'Last;
type Pound is range 0 .. Integer'Last;

Kilograms_1, Kilograms_2 : Kilogram;
Pounds_1,    Pounds_2    : Pound;

...
Kilograms_1 :=  Kilograms_1 + Kilograms_2;  -- STILL LEGAL
Pounds_1    :=  Pounds_1 + Pounds_2;        -- STILL LEGAL
Pounds_1    :=  Kilograms_1 +  Pounds_2;    -- LOGICAL INCONSISTENCY WILL BE
                                            -- CAUGHT BY THE COMPILER
```

INSTRUCTOR NOTES

STRONG TYPING IS ONLY USEFUL WHEN IT IS USED; OTHERWISE THERE ARE NO BENEFITS.

STRONG TYPING

CAN BE USED TO

- CORRECT LARGE CLASS OF ERRORS BEFORE TESTING/INTEGRATION
- DOCUMENT THE DESIGN
- EXPRESS DESIGN CONSTRAINTS OF OBJECTS

INSTRUCTOR NOTES

THIS IS THE LEAD-IN SCENARIO FOR AN EXERCISE IN WHICH THE INSTRUCTOR AND STUDENTS WILL DEFINE TYPES TO REPRESENT THIS DATA. THE INSTRUCTOR SHOULD PUT UP A BLANK FOIL TO DISPLAY AND DISCUSS WITH THE CLASS THE BEST TYPE REPRESENTATIONS FOR EACH TYPE. THE STUDENTS SHOULD NOT GET BOGGED DOWN WITH SYNTAX. THE OBJECTIVE IS TO HAVE THEM IDENTIFY AN APPROPRIATE NAME AND REPRESENTATION. POSSIBLE SOLUTIONS ARE GIVEN IN THE FOLLOWING INSTRUCTOR NOTE.

AIR TRAFFIC CONTROLLER SCENARIO

IN A HYPOTHETICAL, VERY SIMPLIFIED EXAMPLE, AN AIR TRAFFIC CONTROLLER NEEDS TO KEEP TRACK OF SEVERAL ITEMS OF INFORMATION. FOR EACH AIRCRAFT, THESE INCLUDE:

a. THE AIRCRAFT TYPE

b. THE DIRECTION THE PLANE IS TRAVELING

c. THE ALTITUDE OF THE AIRCRAFT

d. THE AIRSPEED OF THE AIRCRAFT

WE ALSO WANT TO CREATE OBJECTS TO CONTAIN

e. ALL THE DATA FOR ONE AIRPLANE

f. THE DATA FOR ALL PLANES IN A SECTOR

a. type Altitude_Type is range 0 .. 40_000;

b. type Aircraft_Type is (L1011, B727, B747, DC10, B767, DC9);

c. type Azimuth is delta 0.01 range 0.0 .. 360.0;

d. type Knots is digits 5 range 0.0 .. 700.0;

e. type Plane_Rec is record
        Altitude : Altitude_Type;
        Aircraft : Aircraft_Type;
        Direction: Azimuth;
        Airspeed : Knots;
    end record;

f. type Control_Sector is array (0 .. 99) of Plane_Rec;

TYPE DECLARATION EXERCISES

DEFINE A TYPE FOR

a. THE ALTITUDE OF A PLANE IF THE MAXIMUM HEIGHT REACHED IS 40,000 FEET

b. THE POSSIBLE AIRCRAFT TYPES, WHICH INCLUDE L1011, B727, B747, DC10, B767, DC9

c. THE DIRECTION OF THE FLIGHT, EXPRESSED IN DEGREES BETWEEN 0 AND 360 TO THE NEAREST HUNDREDTH

d. THE AIRSPEED, WHICH RANGES BETWEEN 0.0 AND 700.0 (EXPRESS TO 5 SIGNIFICANT DIGITS)

e. ALL THE DATA PERTAINING TO A PLANE

f. THE DATA PERTAINING TO ALL THE PLANES IN THE SECTOR (ASSUME 100 MAXIMUM)

HERE THE INSTRUCTOR SHOULD LEAD THEM THROUGH OBJECT DECLARATIONS. AGAIN, SYNTAX SHOULD NOT BE A HINDRANCE. HAVE THEM DETERMINE THE NAME AND TYPE. BELOW ARE POSSIBLE SOLUTIONS.

a. New_Plane : Plane_Rec;

b. Sector : Control_Sector;

OBJECT DECLARATION EXERCISES

NOW THAT WE HAVE THE TYPES (BLUEPRINTS)

DECLARE OBJECTS TO REPRESENT:

a. A PARTICULAR PLANE (E.G., A PLANE THAT HAS NEWLY ENTERED THE SECTOR).

b. ALL THE PLANES IN THE SECTOR

INSTRUCTOR NOTES

SUMMARY

EACH OBJECT MUST BE OF SOME TYPE

OPERATIONS INCOMPATIBLE WITH THE TYPE ARE NOT ALLOWED.

THE TYPE IDENTIFIES THE INTENDED USE.

INSTRUCTOR NOTES

ALLOW 60 MINUTES FOR THIS SECTION. TAKE ABOUT 30 MINUTES FOR THE EXERCISE AT THE END.

SECTION 4
STATEMENT AND CONTROL STRUCTURES

INSTRUCTOR NOTES

STATEMENTS ARE THE OP CODES OF THE HOL MACHINES; THEY "DO" THINGS.

THE LEFT REPRESENTS THE ACTUAL PROGRAM. THE RIGHT REPRESENTS THE ADA MACHINE.

REMEMBER: IN THE ADA MACHINE THE VALUE STORED IS Cold. IN THE ACTUAL MACHINE, THE VALUE STORED IS A BIT PATTERN CHOSEN BY THE COMPILER.

THE LIST OF VALUES IN PARENTHESES IS AN AGGREGATE.

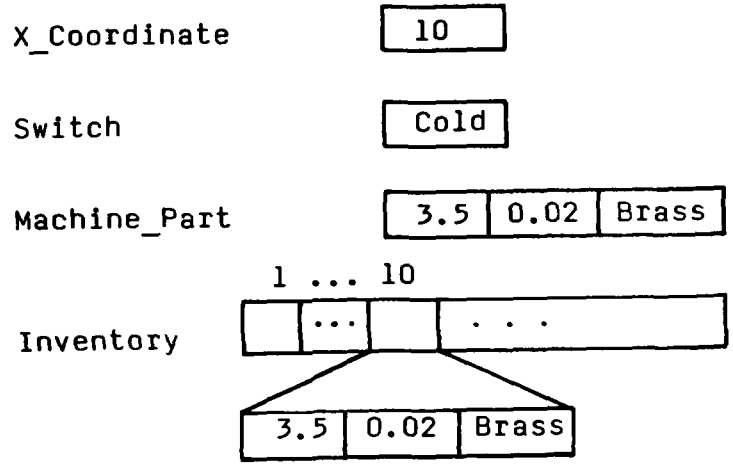
POINT OUT THAT THE EXPRESSION ON THE RIGHT HAND SIDE CAN BE VERY COMPLEX. EXPRESSIONS HAVE NO EQUIVALENT IN ASSEMBLY LANGUAGE.

ASSIGNMENT STATEMENT

CAUSES A VALUE TO BE PLACED IN THE MEMORY LOCATION CORRESPONDING TO A VARIABLE. (IT IS A STORE OPERATION.)

EXAMPLES:

```
X_Coordinate := (35 + 5) / 2 ** 2;

Switch := Cold;

Machine_Part := (3.5, 0.02, Brass);

Inventory (10) := Machine_Part;
```



INSTRUCTOR NOTES

PUT BACK ON THE FOIL ON WHICH YOU WROTE THE SOLUTION TO THE LAST EXERCISE.

LEAD THEM THROUGH THE CREATION OF THE ASSIGNMENT STATEMENTS. AGAIN, ALLOW THEM TO SPECIFY NAMES. DON'T LET SYNTAX BECOME AN ISSUE, ESPECIALLY FOR THE RECORD AND ARRAY.

a. New_Plane := (36_000, DC10, 45.0, 450.0);

b. Sector (1) := New_Plane;

REVISE THE MACHINING EXAMPLE AGAIN; POINT OUT ASSIGNMENTS.

ASSIGNMENT STATEMENT EXERCISES

NOW THAT WE HAVE THE OBJECTS (MEMORY SET ASIDE), ASSIGN THE FOLLOWING:

a. A NEW PLANE TO BE A DC10 FLYING AT 36,000 FEET, 45 DEGREES, AND SPEED 450.0 M.P.H.

f. THE NEW PLANE TO BE IN THE SECTOR AT POSITION 1

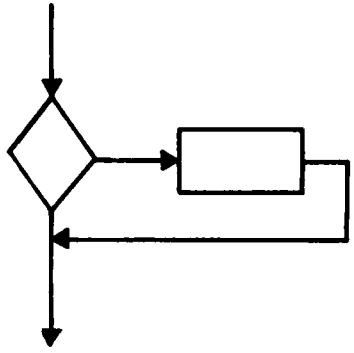
INSTRUCTOR NOTES

THESE CONTROL STRUCTURES CAN BE THOUGHT OF AS THE OP CODES OF THE ADA MACHINE. BY COMBINING THESE, WE CAN INVENT NEW OP CODES.

CONTROL STRUCTURES CORRESPOND TO ALGORITHM PATTERNS. UNLIKE MOST REAL MACHINES, THE HOL MACHINE HAS MANY FORMS OF CONTROL.

EXPLAIN THE FLOW OF CONTROL.

IF-THEN

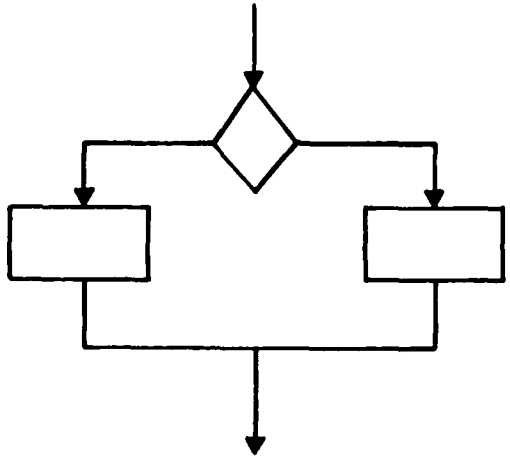


```
if New_Plane_Model = DC9
then
    Max_Airspeed := 600.0;
end if;
```

INSTRUCTOR NOTES

SHOW THE FLOW OF CONTROL. POINT OUT "NO BRANCHES, THE JUMPS ARE CALCULATED BY THE COMPILER."

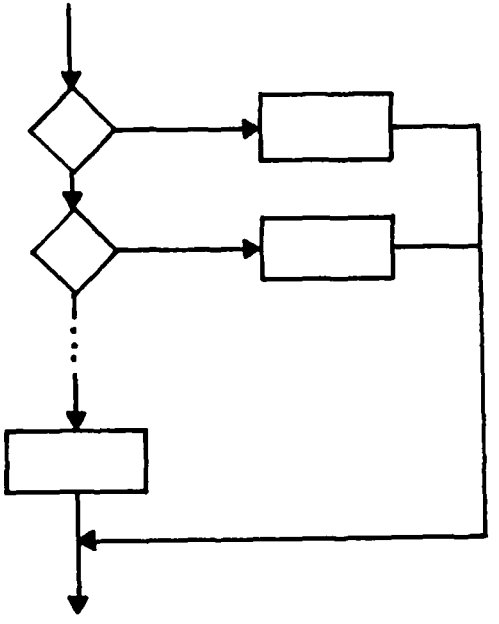
IF-THEN-ELSE



```
if New_Plane_Model = DC9
then
    Max_Airspeed := 600.0;
else
    Max_Airspeed := 700.0;
end if;
```

AGAIN, SHOW THE FLOW OF CONTROL.

IF-THEN-ELSIF

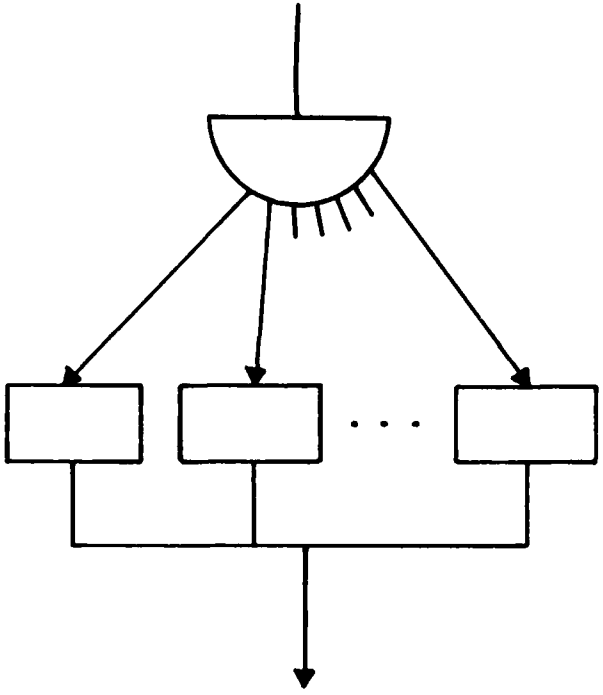


```
if New_Plane_Model = DC9
then
   Max_Airspeed := 600.0;
elsif New_Plane = B727
then
   Max_Airspeed := 625.0;
else
   Max_Airspeed := 700.0;
end if;
```

INSTRUCTOR NOTES

THIS MAY NOT BE THE WAY THE STUDENTS ARE USED TO SEEING FLOW CHARTS. EXPLAIN THE FLOW OF CONTROL FOR BOTH THE FLOW CHART AND THE CASE STATEMENT.

CASE



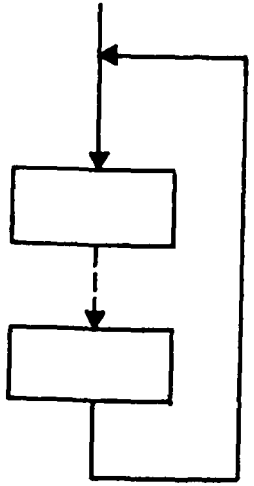
```
case New_Plane_Model is
    when L1011          => Max_Airspeed := 700.0;
    when B747 | DC10    => Max_Airspeed := 685.0;
    when DC9            => Max_Airspeed := 600.0;
    when others         => Max_Airspeed := 625.0;
end case;
```

INSTRUCTOR NOTES

THIS FIRST LOOP IS AN INFINITE LOOP.

ITERATIVE CONTROL STRUCTURES

BASIC LOOP



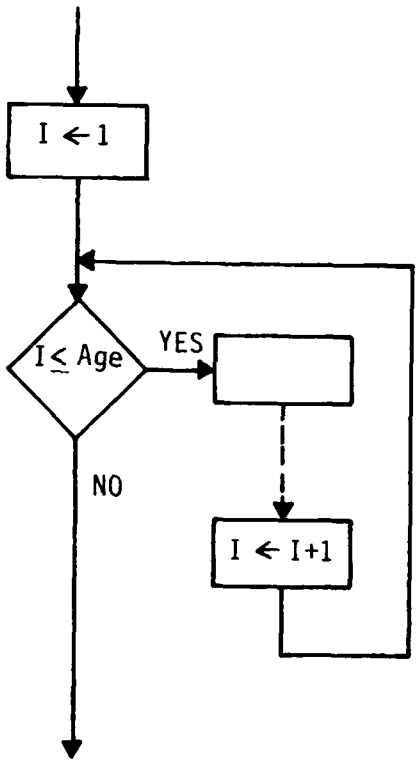
```
loop
   Inhale;
   Exhale;
end loop;
```

INSTRUCTOR NOTES

THE FOR LOOP IS A COUNTED LOOP. NOTE THAT THE INCREMENTING OF THE COUNTER IS DONE AUTOMATICALLY WITH EACH EXECUTION OF THE LOOP.

ITERATIVE CONTROL STRUCTURES

FOR LOOP




```
for I in 1 .. Age
loop
   Light_Birthday_Candle;
end loop;
```

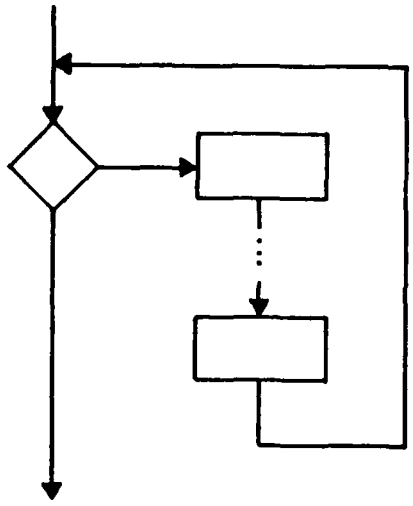
INSTRUCTOR NOTES

NOTE THAT THE BASIC DIFFERENCE BETWEEN THE FOR AND WHILE LOOPS IS THE CONDITION (I.E. REASON) FOR CONTINUING TO LOOP.

ASSUME Alive REPRESENTS A BOOLEAN VALUE.

ITERATIVE CONTROL STRUCTURES

WHILE LOOP



```
while Alive
loop
   Inhale;
   Exhale;
end loop;
```

INSTRUCTOR NOTES

IMPORTANT POINT

MOST COMPUTERS HAVE CONTROL INSTRUCTIONS; THE PROGRAM IS A LINEAR SEQUENCE OF INSTRUCTIONS.

THE ADA MACHINE HAS CONTROL STRUCTURES; THE PROGRAM IS A HIERARCHY OF NESTED CONTROL STRUCTURES.

TOP-DOWN PROGRAMMING MEANS WRITING CONTROL STRUCTURES INSIDE EACH OTHER, RATHER THAN INSTRUCTIONS ONE AFTER THE OTHER.

ON A BLANK FOIL LEAD THE STUDENTS THROUGH WRITING THIS ALGORITHM TOP-DOWN. LET THEM PICK THE CONTROL STRUCTURES AND NAMES TO REPRESENT THE TEMPERATURE. DON'T GET INTO A LONG DISCUSSION ABOUT HOW Get_Temperature, Turn_On_Air_Conditioning, and Turn_Off_Air Conditioning WORK. AFTER THEY HAVE COME UP WITH AN ALGORITHM, YOU MAY WANT TO DISCUSS THIS ONE. DON'T IF THEIRS IS BETTER.

```
loop
        Temperature := Room_Temperature;
        if Temperature > 85.0 then
                Turn_On_Air_Conditioning;
        elsif Temperature < 75.0 then
                Turn_Off_Air_Conditioning;
        end if;
end loop;
```

MENTION IN PASSING THAT THE delay STATEMENT CAN BE USED TO SAMPLE THE TEMPERATURE AT REGULAR INTERVALS.

CONTROL STRUCTURE EXERCISE

WRITE AN ALGORITHM WHICH MONITORS THE TEMPERATURE OF A ROOM AND ADJUSTS THE AIR CONDITIONING DEPENDING ON THE TEMPERATURE. THE PROGRAM SHOULD TURN ON THE AIR CONDITIONING WHEN THE TEMPERATURE RISES ABOVE 85 AND TURN OFF THE AIR CONDITIONING WHEN THE TEMPERATURE DROPS BELOW 75°.

ASSUME: A FUNCTION Room_Temperature EXISTS WHICH RETURNS THE TEMPERATURE OF THE ROOM.

PROCEDURES Turn_On_Air_Conditioning AND Turn_Off_Air_Conditioning EXIST.

TEMPERATURE IS REPRESENTED AS A FLOATING POINT TYPE.

REMINDER: CONTROL STRUCTURE = SEQUENCE, SELECTION (IF, CASE), LOOP

HINT: AT THE OUTERMOST LEVEL, WHAT DO WE HAVE TO DO? A SEQUENCE OF STEPS AND THEN TERMINATE? ONE OF SEVERAL POSSIBLE THINGS? OR SOMETHING REPEATEDLY? CONTINUE TOP-DOWN.

INSTRUCTOR NOTES

ALLOW 75 MINUTES FOR THIS SECTION. TAKE A BREAK AFTER THE FIRST HALF HOUR, WHEN SUBPROGRAMS ARE COMPLETED.

SECTION 5

PROGRAM UNITS

INSTRUCTOR NOTES

"THE STRUCTURING OF PROGRAMS IS IMPORTANT. IT HELPS DETERMINE THE MAINTAINABILITY OF THE PROGRAM."

WHAT DO WE MEAN BY "MODULE"? A SUBROUTINE? A LIBRARY OF MACROS? A MACRO DEFINING DATA LAYOUT? ADA OFFERS A CHOICE OF DIFFERENT KINDS OF MODULES.

PROGRAM UNITS

- SUBPROGRAMS

  PROCEDURES

  FUNCTIONS

- PACKAGES

- TASKS

- GENERIC UNITS

  PROCEDURES

  FUNCTIONS

  PACKAGES

INSTRUCTOR NOTES

TOUCH AGAIN BRIEFLY THE IMPORTANCE OF STRUCTURE AND MAINTAINABILITY.

SUBPROGRAMS

- PROVIDE AN ABSTRACT NAME FOR SOME ALGORITHM OF COMPUTATION
- AVOID DUPLICATION OF CODE EXECUTED IN MORE THAN ONE PLACE
- ENABLE PROGRAMMERS TO MODULARIZE PROGRAMS
- ENHANCE SOFTWARE READABILITY
- ENHANCE SOFTWARE MAINTAINABILITY
- HAVE A RIGOROUS INTERFACE (THE SPECIFICATION)

INSTRUCTOR NOTES

A PROCEDURE IS LIKE A NEW OP-CODE FOR THE ADA MACHINE.

PROCEDURE

- DEFINES A SEQUENCE OF ACTIONS
- IS INVOKED WITH A PROCEDURE CALL STATEMENT

INSTRUCTOR NOTES

POINT OUT THE PROCEDURE CALLS.

TYPICAL PROBLEM

```
with Machine_Shop_Pac;  use Machine_Shop_Pac;  -- contains definitions
procedure Process_Next_Part (Part_Queue  : Queue;
                             Workstation : NC_Machine) is
    This_Part          : Part;
    No_More_Parts      : Boolean;
    Current_Operation  : Operation_ID;
    Tool               : Rotary_Tool_Bit;
    Feed_Rate          : Inch_per_Min;          -- subtype of Float
    Cutting_Speed      : Surface_Ft_per_Min;    --    "    "    "
    Spindle_Speed      : RPM;                   -- subtype of Integer
begin -- Process_Next_Part
    Get_Part (Part_Queue, This_Part, No_More_Parts);
    if No_More_Parts then
        return;
    end if;
    Current_Operation := This_Part.Step (This_Part.Current);
    Load (This_Part, Workstation);
    while Compatible (Current_Operation, Workstation)
    loop
        Tool := Best_Tool (Current_Operation, Workstation);
        Optimize (Cutting_Speed, Feed_Rate,           -- find these
                  Tool.Material, This_Part.Material); -- given these
        Spindle_Speed := RPM(Cutting_Speed /
                             (Pi * Float (Tool.Diameter) / 12.0));
        Set_Up (Workstation, Tool, Spindle_Speed, Feed_Rate);
        Perform_Machining_Step (Current_Operation, Workstation);
    exit when This_Part.Current >= This_Part.Total_Steps;
        This_Part.Current := This_Part.Current + 1;
        Current_Operation := This_Part.Step (This_Part.Current);
    end loop;
    Unload (This_Part, Workstation);
end Process_Next_Part;
```

THE TYPE ARR IS DEFINED TO BE:

```
type Arr is array (Integer range <>) of Float;
```

DON'T GET TOO INVOLVED WITH THE "separate" PROCEDURE. THEY WILL SEE THIS AGAIN IN THE NEXT SECTION, "SEPARATE COMPILATION."

POINT OUT THE PROCEDURE CALL TO Print_Ave.

- THE SPECIFICATION OF Compute_And_Print_Average AS WELL AS OF Print_Ave.

EXPLAIN BRIEFLY ITS STRUCTURE, I.E. DECLARATIONS, begin STATEMENTS.

POINT OUT THE PARAMETERS; THEY WILL HEAR MORE ABOUT THEM IN A FEW SLIDES.

PROCEDURE EXAMPLE

SPECIFICATION

BODY

```
procedure Compute_And_Print_Average (X : in Arr) is

    Sum : Float := 0.0;
    Average : Float;
    procedure Print_Ave (X : Float) is separate;

begin -- Compute_And_Print_Average

    for I in X'Range
    loop
        Sum := Sum + X(I);
    end loop;

    Average := Sum / Float(X'Length);
    Print_Ave (Average);

end Compute_And_Print_Average;
```

LIKE EXPRESSIONS, A FUNCTION HAS NO ASSEMBLER EQUIVALENT.

FUNCTION

- DEFINES A RULE FOR COMPUTING A CERTAIN VALUE
- IS INVOKED WITH A FUNCTION CALL WITHIN AN EXPRESSION
- A FUNCTION CALL CAN APPEAR IN ANY EXPRESSION

INSTRUCTOR NOTES

POINT OUT THE SPECIFICATION, NOTING HOW IT DIFFERS FROM A PROCEDURE.

REVIEW ITS STRUCTURE ALSO.

POINT OUT THE return STATEMENT. EXPLAIN THAT THIS IS NOT OPTIONAL FOR FUNCTIONS.

IF THEY SEEM INTERESTED, WRITE A FUNCTION CALL TO THIS FUNCTION SO THEY CAN SEE HOW IT WOULD BE USED, SUCH AS:

```
Put (Factorial (7));
```

POINT OUT THE PARAMETERS. THEY ARE NEXT.

FUNCTION EXAMPLE

SPECIFICATION

BODY

```
function Factorial (N : Positive) return Positive is
    Product : Positive := 1;
begin -- Factorial
    for I in 1 .. N
    loop
        Product := Product * I;
    end loop;
    return Product;
end Factorial;
```

INSTRUCTOR NOTES

POINT OUT THAT PARAMETERS CAN BE THOUGHT OF AS THE REGISTERS FOR THE HOL MACHINE.

PARAMETERS

PARAMETERS PROVIDE THE MEANS THROUGH WHICH INFORMATION OR DATA IS TRANSFERRED BETWEEN THE CALLED SUBPROGRAM AND ITS CALLER. THIS TRANSFER MAY WORK IN THE FOLLOWING DIRECTIONS:

CALLER --> CALLED

CALLER <-- CALLED

CALLER <--> CALLED

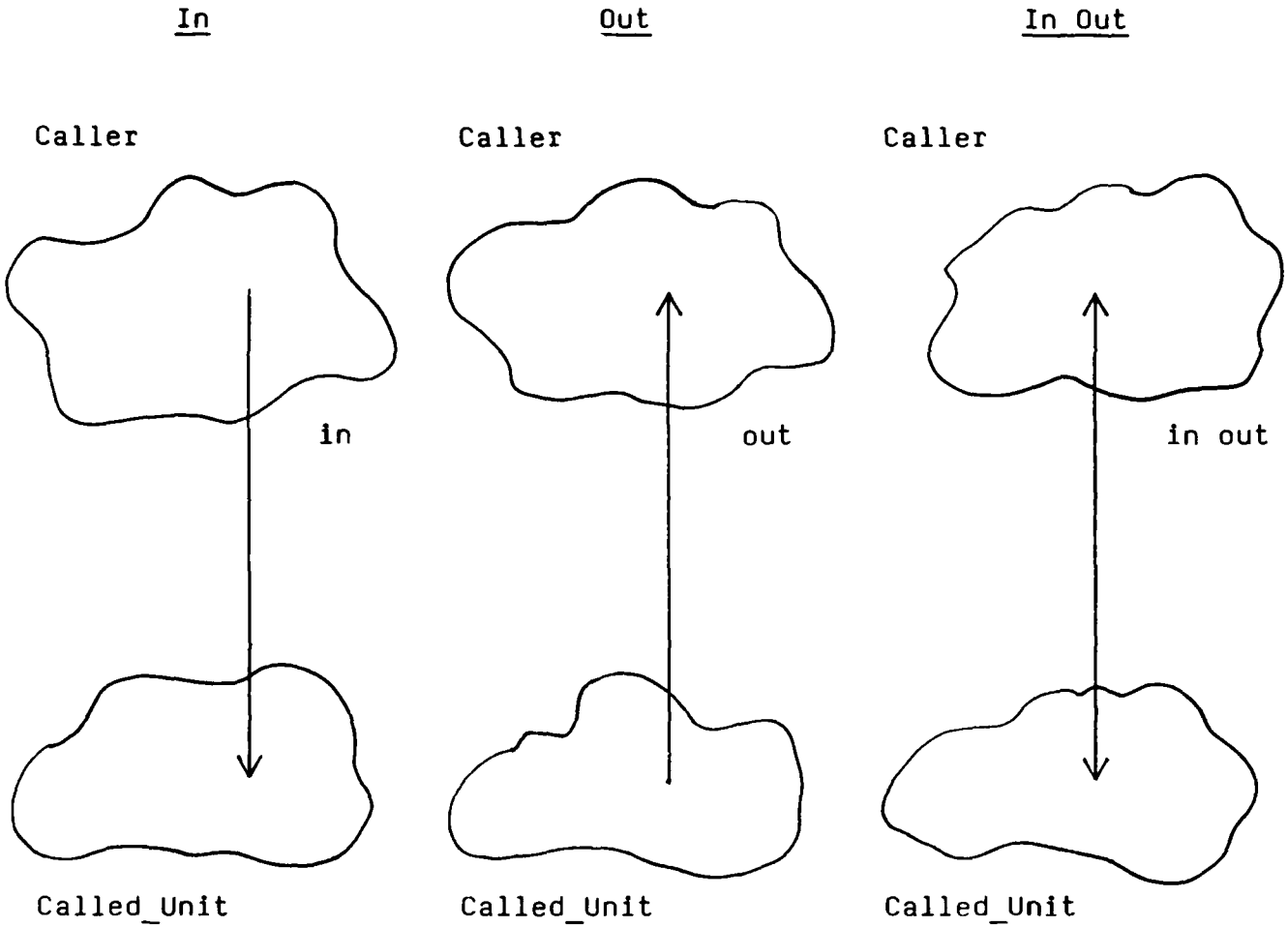
THE DIRECTION IS SPECIFIED BY THE MODE.

THE PARAMETERS OF PROCEDURES CAN HAVE ANY MODE.

THE PARAMETERS OF FUNCTIONS MUST BE OF MODE in.

WHEN NO MODE IS EXPLICITLY STATED, "IN" IS ASSUMED.

PARAMETER MODES



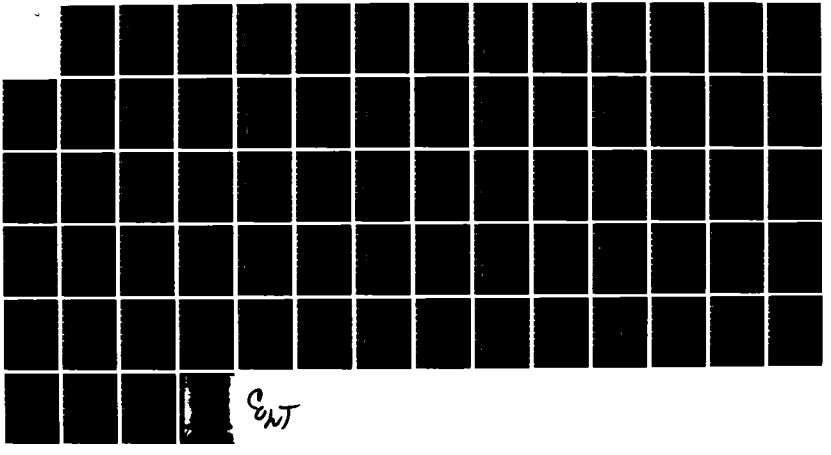
INSTRUCTOR NOTES

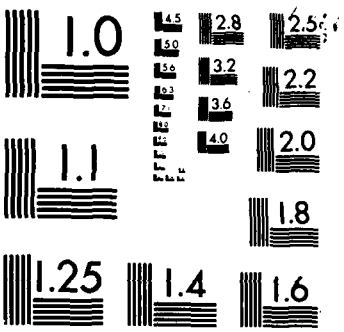
TYPICALLY A PROGRAM'S STRUCTURE APPEARS AS NESTED UNITS, AS SHOWN BELOW.

NOTE THAT ALL PROGRAM UNITS CAN BE NESTED. ONLY SUBPROGRAMS ARE SHOWN BELOW.

AD-A141 848 INTRODUCTION TO ADA A HIGHER ORDER LANGUAGE L103 TEACHER'S GUIDE(U) SOFTECH INC WALTHAM MA MAY 84 DAAB07-83-C-K514 3/4

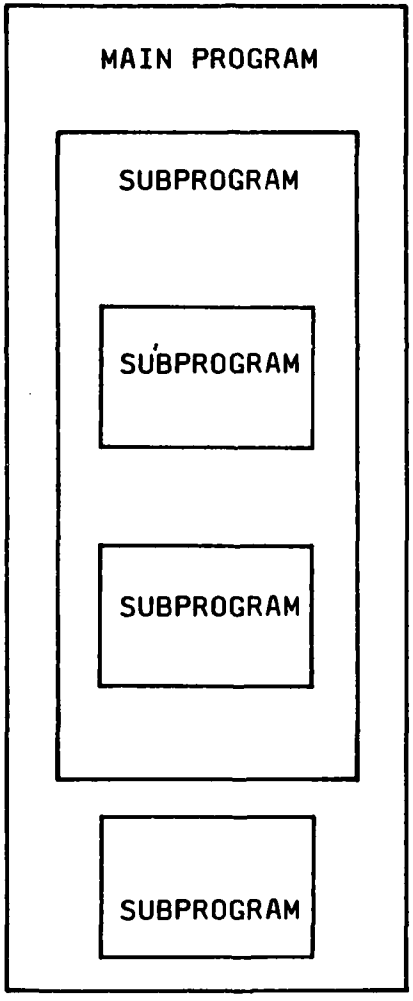
UNCLASSIFIED F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

NESTING OF SUBPROGRAMS



POINT OUT THE NESTED SUBPROGRAMS. THE ... ARE NOT A PROGRAM STRUCTURE; THEY MEAN, "STUFF HAS BEEN LEFT OUT HERE."

NOTE HOW FUNCTION CALL DEFINES A VALUE, PROCEDURE CALL DEFINES AN ACTION.

EXAMPLE OF NESTING

```
function Header_Valid (Message: Message_Type) return Boolean is

    type Validity is (Valid, Bad_RI, Bad_LMF, Security_Mismatch);
    Result: Validity;

    function Security_Validity (Message: in Message_Type) return Validity is
        . . .
    begin           -- Security_Validity
        . . .
    end Security_Validity;

    procedure Reject_Message (Message: Message_Type; Error: Validity_Type) is
        . . .
    begin           -- Reject Message
        . . .
    end Reject_Message;

begin               -- Header_Valid

    Result := Security_Validity (Message);
    if Result /= Valid
    then
        Reject_Message (Message, Result);
        return False;
    else
        return True;
    end if;

end Header_Valid;
```

INSTRUCTOR NOTES

TAKE A 15-MINUTE BREAK BEFORE CONTINUING. ALLOW 45 MINUTES FOR THE REST OF THIS SECTION. SPEND 20 MINUTES ON PACKAGES, 15 MINUTES ON TASKS, AND 10 MINUTES ON GENERICS.

POINT OUT THAT PACKAGES HAVE NOTHING TO DO WITH THE INTERNAL RUNTIME BEHAVIOR. STRESS THAT THEY ARE EXTERNAL STRUCTURING MECHANISMS.

STUDENTS SHOULD BE ALREADY FAMILIAR WITH THE CONCEPT OF "PACKAGE" OF ROUTINES (OR MACROS).

PACKAGES

- ARE BASIC STRUCTURING MECHANISMS TO AID DEVELOPMENT AND MAINTENANCE
- GROUP FUNCTIONALLY RELATED DATA AND PROGRAM UNITS
- ARE STRUCTURE REPRESENTATIONS, NOT ALGORITHMS
- PROVIDE FOR REUSABLE SOFTWARE COMPONENTS, GROUPED TO BENEFIT OTHER PROGRAM UNITS
- INCREASE MAINTAINABILITY BECAUSE EFFECT OF CHANGES CAN BE LOCALIZED
- ARE COMPOSED OF SPECIFICATION AND BODY

INSTRUCTOR NOTES

PACKAGES WHICH GROUP ONLY TYPES AND OBJECTS DO NOT NEED A PACKAGE BODY.

EXAMPLE PACKAGE SPECIFICATION

```
package Database is

    type Data_Entry is
        record
            Name  : String (1 .. 20);
            Sex   : Boolean;
            Age   : Integer;
        end record;

    type Data_Records is array (1 .. 20) of Data_Entry;

    Cur_Database : Data_Records;

end Database;        -- WILL NEED NO PACKAGE BODY
```

INSTRUCTOR NOTES

NOTE THAT FOR PACKAGE THE SPECIFICATION ALWAYS IS SEPARATE FROM THE BODY. THE SPECIFICATION IS THE INTERFACE.

IN THIS EXAMPLE, DON'T GO INTO TOO MUCH DETAIL. TRY TO AVOID EXPLAINING THE EXCEPTIONS, EXCEPT TO SAY THAT THIS IS A WAY OF SIGNALLING ERROR SITUATIONS. THEY WILL SEE MORE OF THESE IN LATER MODULES.

EXAMPLE PACKAGE SPECIFICATION AND BODY

SPECIFICATION

```
package Stack_Manager is

        procedure Push (X: Integer);
        function Pop return Integer;
        Stack_Full, Stack_Empty : exception;

end Stack_Manager;
```

BODY

```
package body Stack_Manager is

        type Stack is array (1 .. 100) of Integer;
        Top : Integer := 1;
        Int_Stack : Stack;

        procedure Push (X: Integer) is
        begin -- Push

                if Top > Stack'Last then
                        raise Stack_Full;
                else
                        Int_Stack (Top) := X;
                        Top := Top + 1;
                end if;

        end Push;

        function Pop return Integer is
        begin -- Pop

                if Top = Stack'First then
                        raise Stack_Empty;
                else
                        Top := Top - 1;
                        return Int_Stack (Top);
                end if;

        end Pop;

end Stack_Manager;
```

INSTRUCTOR NOTES

THIS MAY BE A VERY CONFUSING AREA FOR ASSEMBLY LANGUAGE PROGRAMMERS. EXPLAIN THAT THIS IS A VERY ADVANCED CONCEPT THAT THEY NEED ONLY BE AWARE OF.

"THE ADA MACHINE CAN BE THOUGHT OF AS SEVERAL MACHINES THAT OPERATE IN PARALLEL. TASKS ARE THE PROGRAM UNITS THAT THESE 'MACHINES' EXECUTE."

TASKS

- PARALLEL THREADS OF CONTROL
- EXECUTION IS CONCURRENT WITH OTHER TASKS
- SYNCHRONIZATION WITH OTHER TASK POSSIBLE
- DATA CAN BE PASSED TO OTHER TASK
- MECHANISM FOR SYNCHRONIZATION AND DATA TRANSMISSION IS CALLED "RENDEZVOUS"
- DIRECT MAPPING OF REALTIME PROCESSING DESIGNS INTO THE LANGUAGE

INSTRUCTOR NOTES

THE DETAILS OF TASKS WILL NOT BE DISCUSSED IN THIS MODULE.

TASKS

CAN BE USED FOR

- CONCURRENT ACTIONS
- CONTROLLING RESOURCES
- HANDLING INTERRUPTS
- CONTROLLING BUFFERS

INSTRUCTOR NOTES

QUICKLY GO OVER THIS EXAMPLE. THE MAJOR POINT IS TO UNDERSTAND THE SYNCHRONIZATION.

TASKING EXAMPLE

MAIN PROGRAM

```
with Text_IO;
procedure Main is
    type Card is String (1 .. 80);
    C : Card;

    task Card_Reader is
         entry Get (C : out Card);
    end Card_Reader;

    task body Card_Reader is separate;

begin -- Main
    loop
         Card_Reader.Get (C);
         Text_IO.Put (C);
    end loop;

end Main;
```

SPECIFICATION

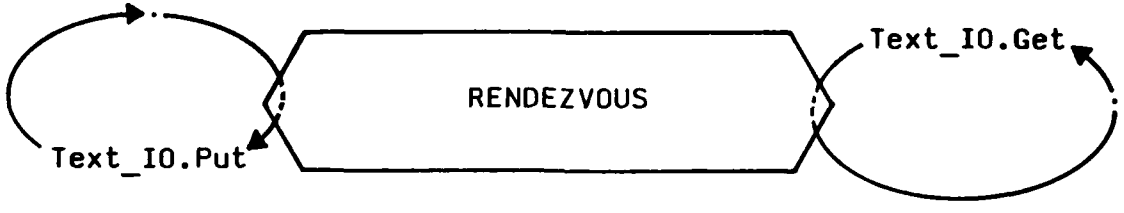
CARD READER TASK

```
separate (Main)

task body Card_Reader is
     Latest_Card : Card;

begin -- Card_Reader
     loop
          Text_IO.Get (Latest_Card);
          accept Get (C : out Card) do
               C := Latest_Card;
          end Get;
     end loop;

end Card_Reader;
```

BODY

ACCEPT

ENTRY CALL



INSTRUCTOR NOTES

NOTE THAT THE ANALOGY OF GENERICS TO MACROS IS RATHER SHALLOW. TELL THEM TO BEWARE OF THIS.

IF THEY SEEM RECEPTIVE (OR IF YOU NEED TO TELL THEM SOMETHING GOOD ABOUT GENERICS), TELL THEM THAT GENERICS CAN BE MORE EFFICIENT THAN MACROS. MACROS ARE LITERAL CODE INSERTIONS, SO IN SOME CASES, DEAD CODE MAY EXIST. A COMPILER COULD DELETE THIS FROM A GENERIC. ALSO, GENERICS MAY HAVE BODY SHARING.

GENERIC PROGRAM UNITS - BASIC IDEA

- PROVIDE EFFECTIVE REUSE OF COMMON BUT SLIGHTLY DIFFERENT CODE
- ARE PARAMETERIZED TEMPLATES OF A PROGRAM UNIT
- WRITE CODE TO BE SLIGHTLY MODIFIED AND REUSED
- CAN BE THOUGHT OF AS A MACRO WITH A RIGOROUS INTERFACE
- "INSTANTIATION" CREATES AN EXECUTABLE COPY OF THE PROGRAM UNIT AND SUBSTITUTES THE PARAMETERS
- GENERIC UNITS CAN BE PROCEDURES, FUNCTIONS, AND PACKAGES

EXAMPLE: SORTING A LIST OF CHARACTERS VS. SORTING A LIST OF INTEGERS. THE LOGIC IS THE SAME, BUT THE TYPES ARE DIFFERENT. IN ASSEMBLER, DIFFERENT OP CODES WOULD HAVE TO BE USED HERE AND THERE; IN ADA (WITHOUT GENERICS) DIFFERENT DECLARATIONS WOULD HAVE TO BE WRITTEN.

THE LAST POINT MEANS THAT SYSTEMS USING GENERICS CAN BE TESTED AND DEBUGGED USING UNCLASSIFIED INFORMATION. IT'S NOT ONE OF THE TYPICAL APPLICATIONS OF GENERICS, BUT SHOWS THEIR VERSATILITY.

GENERICS

CAN BE USED TO

- EXPLOIT SIMILARITY OF PROGRAM UNITS DIFFERING ONLY IN SPECIFIC TYPES OR IN MINOR COMPUTATIONAL DETAILS
- DETAIL DESIGN OPTIONS
- DEFER DESIGN DECISIONS
- (DEVELOP AND TEST SOFTWARE OF A CLASSIFIED NATURE)

INSTRUCTOR NOTES

THE EXAMPLE IS HERE SO THEY CAN SEE THE FORM. DON'T SPEND A LOT OF TIME GOING OVER IT. (YOU MAY EVEN SKIP IT IF SHORT ON TIME.) DO POINT OUT THE INSTANTIATIONS.

GENERIC UNITS EXAMPLE

TEMPLATE:

```
generic -- GENERIC CLAUSE
        type Swap_Type is private;
procedure Exchange (X, Y: in out Swap_Type);

procedure Exchange (X, Y: in out Swap_Type) is
        Temp : Swap_Type := X;
begin -- Exchange
        X := Y;
        Y := Temp;
end Exchange;
```

INSTANTIATIONS:

```
procedure Swap is new Exchange (Integer);
procedure Swap is new Exchange (Character);
procedure Swap is new Exchange (Day);
procedure Swap is new Exchange (String (1 .. 10));
```

INSTRUCTOR NOTES

SCOPE AND VISIBILITY

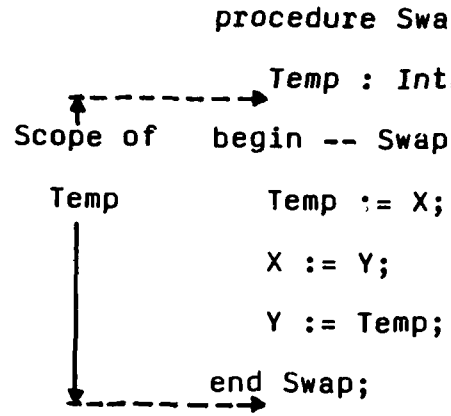
EVERY PROGRAM UNIT (MODULE) SHOULD BE ABLE TO DECLARE THE RESOURCES IT NEEDS (DATA, SUBROUTINES, ETC.) WITHOUT INTERFERING WITH OTHER PROGRAM UNITS.

- AVOID UNINTENDED NAME CONFLICTS
- MAINTAIN TOTAL CONTROL OVER LOCAL RESOURCES. PREVENT UNAUTHORIZED ACCESS (VARIABLE "CLOBBERING;" UNDOCUMENTED CALLS; ETC.)
- ALLOW INTENDED COMMUNICATION VIA DECLARED INTERFACES

INSTRUCTOR NOTES

BASIC IDEA

CONSIDER A VARIABLE DECLARED IN A PROCEDURE:



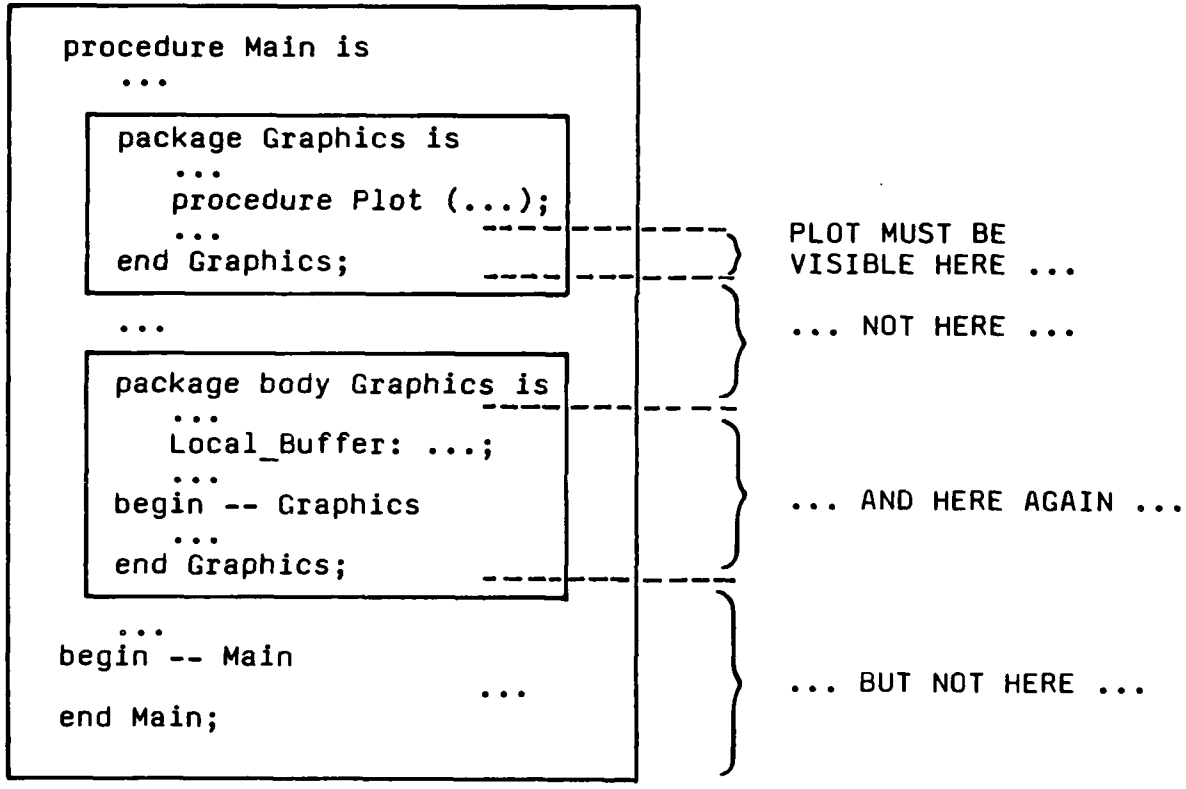
1. Temp IS NOT VISIBLE (I.E., CANNOT BE MENTIONED) OUTSIDE ITS SCOPE.

2. Temp COMES INTO EXISTENCE AT THE BEGINNING OF ITS SCOPE AND CEASES TO EXIST AT THE END. (AT LEAST CONCEPTUALLY, A STACK IS USED FOR LOCAL DATA).

INSTRUCTOR NOTES

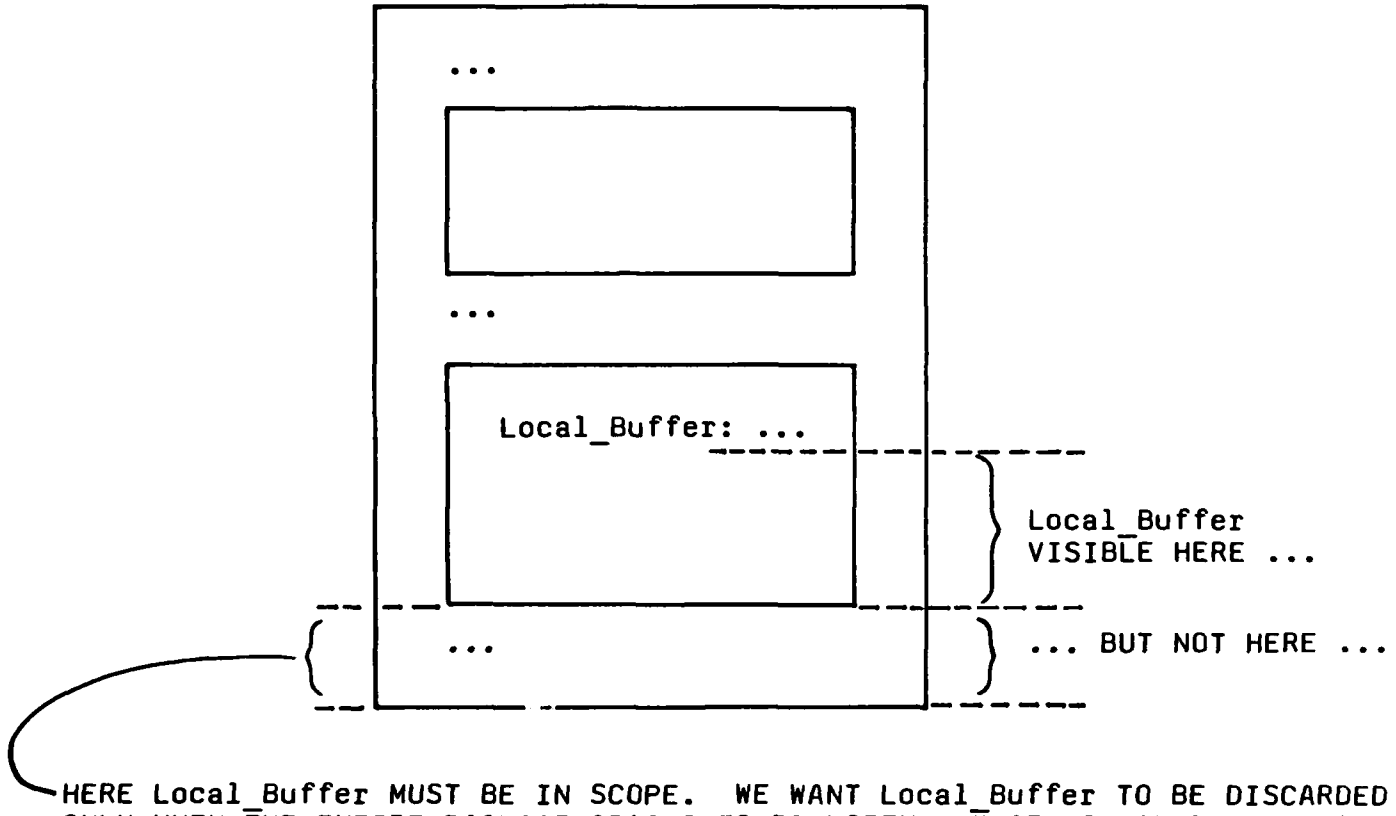
BASIC IDEA (CONTINUED)

THINGS ARE MORE COMPLICATED FOR A PACKAGE.



INSTRUCTOR NOTES

BASIC IDEA (CONTINUED)



INSTRUCTOR NOTES

THERE WILL BE MORE ON NAME CONFLICTS IN L202.

PURPOSE OF SCOPE AND VISIBILITY RULES

- SCOPE RULES CONTROL THE LIFETIME OF ENTITIES

  FOR EXAMPLE:

  WHEN STORAGE CAN BE RECLAIMED

- VISIBILITY RULES PREVENT ACCIDENTAL NAME CONFLICTS

  FOR EXAMPLE:

  DIFFERENT SUBPROGRAMS CAN HAVE "LOCAL" VARIABLES WITH THE SAME NAME

INSTRUCTOR NOTES

- "DON'T PANIC. EXAMPLE FOLLOWS."
- DECLARATIONS WITH EXTENDED SCOPE INCLUDE:
  - DECLARATIONS IN THE VISIBLE PART OF A PACKAGE SPECIFICATION. (THIS ALLOWS THEM TO BE USED THROUGHOUT THE SCOPE OF THE ENCLOSING PACKAGE.) BE CAREFUL HERE. THEY DON'T KNOW "VISIBLE PART."
  - SUBPROGRAM PARAMETERS (SO THEY CAN BE USED IN NAMED PARAMETER LISTS ANYWHERE THE SUBPROGRAM CAN BE CALLED)
  - RECORD COMPONENTS (SO THEY CAN BE ACCESSED ANYWHERE THE RECORD CAN BE)
- NOTE THAT DECLARATIONS WITH EXTENDED SCOPE ARE IN A SENSE PROPERTIES OF THE PARENT, SO IT IS REASONABLE THAT THEY HAVE THE SAME SCOPE AS THE PARENT.

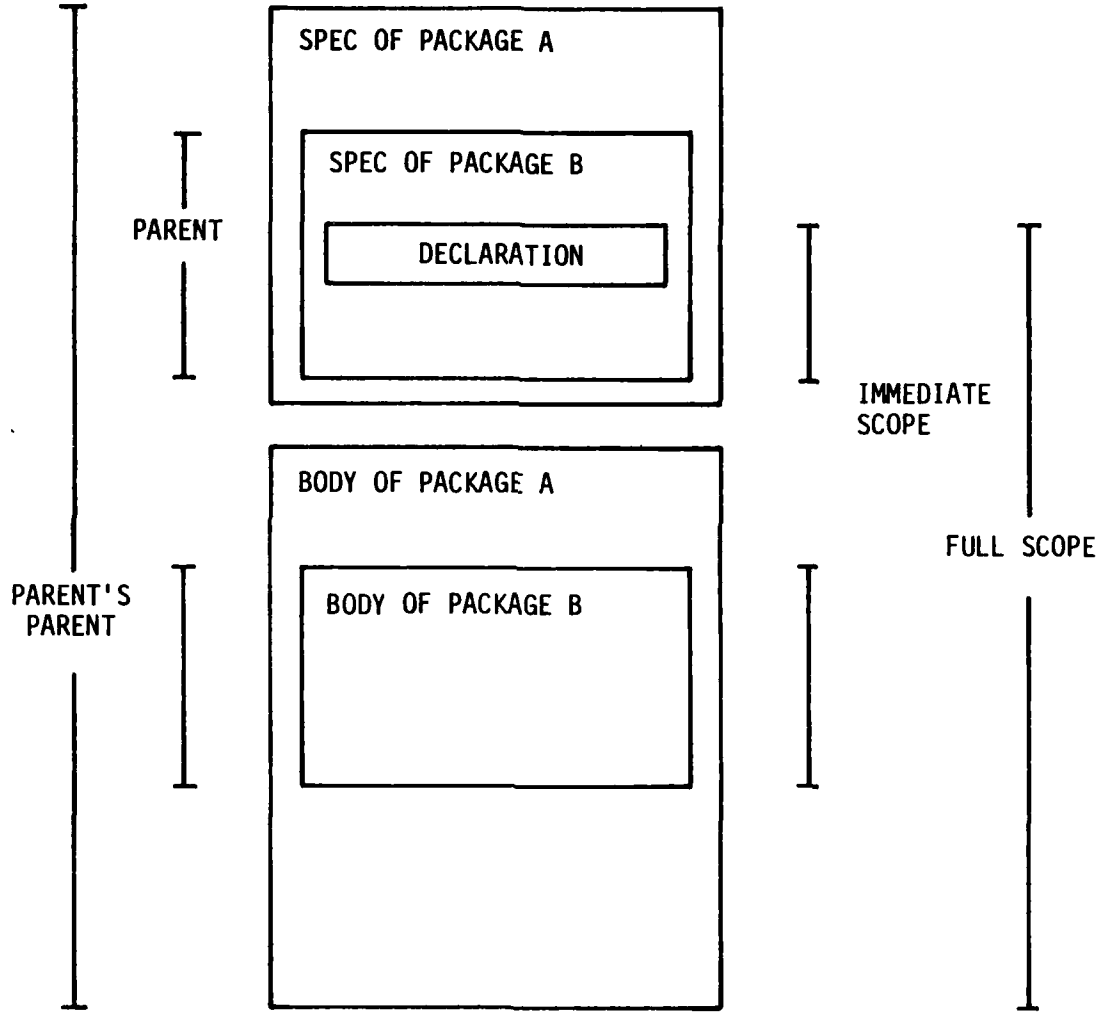
SCOPE AND VISIBILITY: DEFINITIONS

- ALL DECLARED ITEMS HAVE A SCOPE.
- THE SCOPE OF A DECLARATION IS THE REGION OF PROGRAM TEXT IN WHICH THE ENTITY EXISTS.
- THE IMMEDIATE SCOPE OF A DECLARATION EXTENDS FROM THE DECLARATION TO THE END OF THE IMMEDIATELY ENCLOSING DECLARATIVE REGION. THIS IMMEDIATELY ENCLOSING REGION IS REFERRED TO AS THE PARENT OF THE DECLARATION.
- HOWEVER, SOME DECLARATIONS HAVE AN EXTENDED SCOPE THAT REACHES TO THE END OF THE FULL SCOPE OF THE PARENT.
- THE FULL SCOPE OF A DECLARATION IS THE IMMEDIATE SCOPE PLUS THE EXTENDED SCOPE (IF ANY).
- THE RELATED CONCEPT OF VISIBILITY DETERMINES WHERE WITHIN THE SCOPE THE IDENTIFIER WILL BE RECOGNIZED AS REFERRING TO THE DECLARED ENTITY.

INSTRUCTOR NOTES

NOTICE THE NESTING OF PACKAGES. SPECIFICATIONS ARE NESTED IN SPECIFICATIONS (OR BODIES) AND BODIES ARE NESTED IN BODIES. BODIES ARE NEVER NESTED IN PACKAGE SPECIFICATIONS.

SCOPE - ILLUSTRATION



INSTRUCTOR NOTES

SCOPE - WRAP-UP

- ALTHOUGH THE SCOPE RULES ARE COMPLICATED TO STATE, THEIR IMPLICATIONS ARE STRAIGHTFORWARD:

  - NORMAL DECLARATIONS HAVE A SCOPE THAT INCLUDES ONLY THE CONTAINING REGION (THEY CANNOT BE ACCESSED OUTSIDE OF THAT UNIT)

  - DECLARATIONS VISIBLE OUTSIDE THE PARENT (VISIBLE PACKAGE DECLARATIONS, SUBPROGRAM PARAMETERS, RECORD COMPONENTS) HAVE THE SAME SCOPE AS THE PARENT

- NOTE THAT THE SCOPE RULES ARE RECURSIVE, SO THAT IF THE PARENT IS VISIBLE OUTSIDE ITS PARENT, VISIBLE DECLARATIONS IN INNER REGIONS WILL HAVE SCOPE INCLUDING THE PARENT'S PARENT'S PARENT, ETC.

INSTRUCTOR NOTES

ALLOW 30 MINUTES FOR THIS SECTION.

SECTION 6

SEPARATE COMPILATION

INSTRUCTOR NOTES

BE AWARE THAT WORKING IN A TEAM ENVIRONMENT MAY BE VERY FOREIGN TO THE STUDENTS.

EXPLAIN THAT THE SPECIFICATION OF A SUBPROGRAM CAN BE SEPARATE, EVEN THOUGH THEY HAVEN'T SEEN IT SO FAR. IF YOU WANT, HAVE THEM LOOK BACK AT THE EXAMPLE OF THE PACKAGE SPECIFICATION AND BODY. AT LEAST SHOW SUBPROGRAM SPECIFICATIONS SEPARATE.

SEPARATE COMPILATION

- ALLOWS MANY PROGRAMMERS TO BE DEVELOPING A SYSTEM CONCURRENTLY
- DIFFERENT PROGRAM UNITS CAN BE COMPILED SEPARATELY
- THE BODY OF A PROGRAM CAN BE COMPILED SEPARATELY FROM ITS CORRESPONDING SPECIFICATION

INSTRUCTOR NOTES

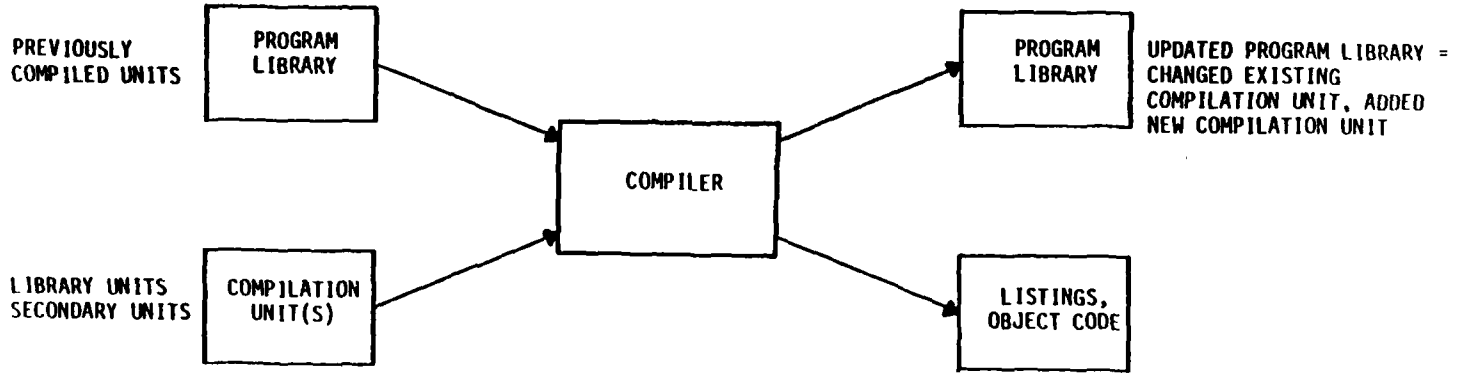
THE NEXT SLIDE EXPLAINS COMPILATION UNITS, AND THE ONE AFTER THAT EXPLAINS PROGRAM LIBRARIES.

AN ADA PROGRAM

CONSISTS OF A COLLECTION OF ONE OR MORE "COMPILATION" UNITS THAT CAN BE SUBMITTED TO A COMPILER ALL TOGETHER OR IN SEVERAL SEPARATE RUNS.

ONCE COMPILED, THESE "COMPILATION" UNITS FORM A PROGRAM LIBRARY.

PICTORIALLY,



INSTRUCTOR NOTES

COMPILATION UNITS

- LIBRARY UNITS
  - PACKAGE SPECIFICATION
  - GENERIC SPECIFICATION
  - GENERIC INSTANTIATION
  - SUBPROGRAM SPECIFICATION
  - SUBPROGRAM BODY
- SECONDARY UNITS
  - PACKAGE BODY
  - SUBPROGRAM BODY
  - SUBUNITS (BODY STUBS)

INSTRUCTOR NOTES

PROGRAM LIBRARY

- EVERY ADA PROGRAM HAS AN ASSOCIATED PROGRAM LIBRARY.

- AS LIBRARY UNITS ARE COMPILED, THEIR DECLARATIONS GO INTO THE PROGRAM LIBRARY.

- THE COMPILER USES THE PROGRAM LIBRARY TO PROCESS A REFERENCE TO A PREVIOUSLY COMPILED LIBRARY UNIT.

INSTRUCTOR NOTES

COMPILATION AND LARGE SYSTEMS

IN DEVELOPMENT OF LARGE SYSTEMS

- PROGRAMMER NEEDED TO WORK CONCURRENTLY
- NEED MECHANISMS FOR TOP-DOWN AND BOTTOM-UP DEVELOPMENT
  - TOP-DOWN
    - LARGE COHERENT PROGRAM BROKEN DOWN INTO SEPARATELY-COMPILED SUBUNITS
    - SUBUNITS COMPILED AFTER UNIT ON WHICH THEY DEPEND
    - MECHANISM IMPLEMENTED USING "...is separate" AND "separate ..." NOTATION (BODY STUB)
  - BOTTOM-UP
    - TYPICAL APPLICATION IS A "LIBRARY" OF SUBPROGRAMS WRITTEN FOR GENERAL USE
    - THESE SUBPROGRAMS ARE WRITTEN BEFORE UNITS THAT USE THEM
    - UNITS THAT USE SUCH SUBPROGRAMS (I.E., THAT DEPEND ON THEM) GET ACCESS VIA with CLAUSES

THIS IS THE "is separate" STUFF THEY SAW EARLIER.

TOP-DOWN DEVELOPMENT

- BREAKING AN ADA PROGRAM INTO SEPARATELY COMPILED LIBRARY UNITS

- USING BODY STUBS AND SUBUNITS

- AT THE POINT WHERE A SUBPROGRAM BODY OR PACKAGE BODY WOULD NORMALLY APPEAR IN A COMPILATION, A BODY STUB MAY BE USED INSTEAD

```
procedure Subprogram_Name is separate;
```

  THIS IMPLIES THAT THE ACTUAL BODY WILL BE SUPPLIED IN A SEPARATE SUBUNIT

- ALTHOUGH THE SUBUNIT IS SEPARATELY COMPILED THE EFFECT IS EXACTLY AS IF THE ACTUAL BODY WERE GIVEN AT THE POINT OF THE BODY STUB

- ALL DECLARATION VISIBILITY IS THE SAME AS AT THE POSITION OF THE BODY STUB

INSTRUCTOR NOTES

STUBBING

CAN BE USED TO

- INCREASE UNDERSTANDABILITY OF A DESIGN
- INCREASE UNDERSTANDABILITY OF COMPLEX CODE
- LOCALIZE LIBRARY UNIT INFORMATION
- DECREASE RECOMPILATION COSTS

INSTRUCTOR NOTES

STUBBING NOTE

SUBUNITS ONLY ELIMINATE PHYSICAL NESTING OR COMPLEXITY, BUT DO NOTHING FOR LOGICAL COMPLEXITY.

POINT OUT THAT ANY OF THE POINTS THEY ARE NOT FAMILIAR WITH WILL BE COVERED IN OTHER MODULES. (THEY SHOULD HAVE HAD IT IN M102.)

BOTTOM-UP DEVELOPMENT

- BEGIN WITH AN IDENTIFICATION OF LIBRARY PACKAGES
- THE CRITERIA TO BE USED IN IDENTIFYING LIBRARY PACKAGES INCLUDE:
  - ACCESS TO COMMON DATA
  - FUNCTIONAL COHESIVENESS
  - SOFTWARE DEVELOPMENT TEAM STRUCTURE

THE with CLAUSE IS NOT A RUNTIME ISSUE. ITS EFFECT WHEN COMPILED IS TO MAKE SURE INTERFACES MATCH AND ARE COMPATIBLE AND TO LEAVE INSTRUCTIONS FOR THE LOADER .

WITH CLAUSES

- TO USE A SEPARATELY COMPILED PROGRAM UNIT IN ANOTHER SEPARATELY COMPILED PROGRAM UNIT, THE SECOND UNIT MUST CONTAIN A with CLAUSE SPECIFYING THE NAME OF THE PROGRAM UNIT BEING USED

  with Library_Unit_Name, ..., Library_Unit_Names;

- THE with CLAUSE MAKES THE DECLARATIONS IN THE VISIBLE PART OF THE NAMED PACKAGE VISIBLE BY SELECTION IN THE PROGRAM UNIT.

- THE with CLAUSE MUST APPEAR AT THE BEGINNING OF THE UNIT BEING COMPILED.

- WHY DO YOU NEED THE with CLAUSE? SO THAT READERS KNOW UP FRONT THE CONNECTIONS TO OTHER MODULES AND SO TYPOS CAN BE DETECTED (SIMILAR TO A DECLARATION, IN THIS RESPECT). SO THAT THE COMPILER KNOWS THAT YOU ARE REFERRING TO A PREVIOUSLY COMPILED LIBRARY UNIT, WHOSE DECLARATION WILL BE FOUND IN THE PROGRAM LIBRARY.

```
with Text_IO; -- declares procedures Get and Put
procedure Transfer is
       Line : String (1 .. 80);
begin -- Transfer
       Text_IO.Get (Line);
       Text_IO.Put (Line);
end Transfer;
```

INSTRUCTOR NOTES

THIS IS NOT TOO IMPORTANT. SKIP IF SHORT OF TIME.

COMPILATION ORDER

- A COMPILATION UNIT MUST BE COMPILED AFTER ALL LIBRARY UNITS NAMED BY ITS with CLAUSES.

- A SECONDARY UNIT (A SUBPROGRAM OR PACKAGE BODY) MUST BE COMPILED AFTER THE CORRESPONDING SUBPROGRAM OR PACKAGE SPECIFICATION.

- A SUBUNIT (separate STUB) MUST BE COMPILED AFTER ITS PARENT COMPILATION UNIT.

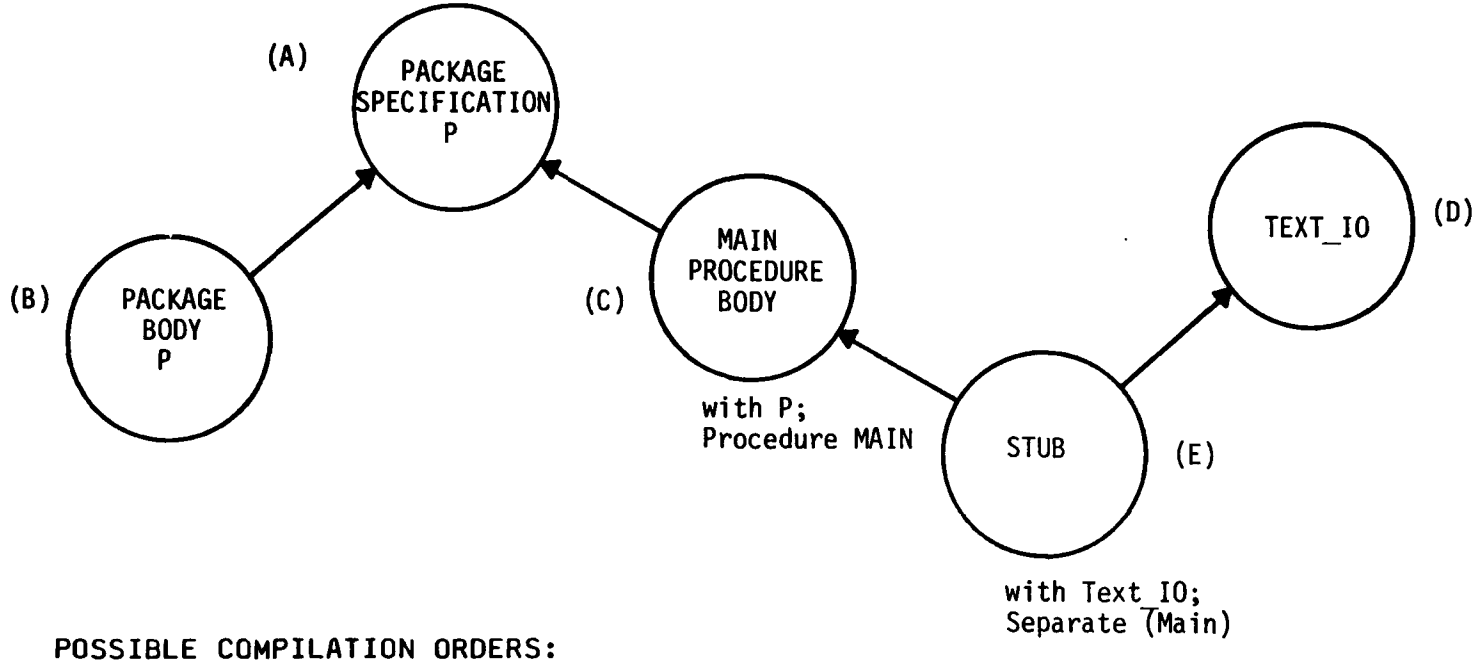
AS A RULE:

A GIVEN UNIT MUST BE COMPILED AFTER ANY UNITS CONTAINING INFORMATION UPON WHICH IT DEPENDS.

- IF UNIT A MUST BE COMPILED BEFORE UNIT B, THEN IF A IS RECOMPILED B MUST BE RECOMPILED (UNLESS A SMART COMPILER CAN DETERMINE THAT ANY CHANGES MADE TO A DO NOT AFFECT B).

INSTRUCTOR NOTES

COMPILATION ORDER DEPENDENCIES



POSSIBLE COMPILATION ORDERS:

(A) (B) (C) (D) (E)
(A) (C) (B) (D) (E)
(A) (C) (D) (B) (E)
(A) (B) (D) (C) (E)
(A) (D) (B) (C) (E)
(A) (D) (C) (B) (E)
(A) (C) (D) (E) (B)
(A) (D) (C) (E) (B)

(D) (A) (B) (C) (E)
(D) (A) (C) (B) (E)
(D) (A) (C) (E) (B)

*ACTUALLY, (D) -- Text_IO -- IS "PRE-COMPILED," SO THE POSSIBILITIES WHERE ONLY D IS DIFFERENT DOES NOT ARISE.

INSTRUCTOR NOTES

SHOULD HAVE 30 MINUTES FOR THIS SECTION (IF OTHER TIMINGS WERE FOLLOWED). A MAJOR PORTION OF THIS TIME SHOULD BE SPENT ANSWERING QUESTIONS.

SECTION 7

SUMMARY

INSTRUCTOR NOTES

CHARACTERISTICS OF THE ADA MACHINE

- TYPES (INCLUDING COMPOSITE TYPES)
- OBJECTS; AUTOMATIC STORAGE MANAGEMENT
- EXPRESSIONS
- STATEMENTS
- CONTROL STRUCTURES
- SUBPROGRAMS: PARAMETERS
- TASKS

INSTRUCTOR NOTES

MANY FEATURES EXIST TO AID DEVELOPMENT AND MAINTENANCE.

REMIND THEM THAT THEY WILL SEE ALL OF THESE FEATURES AGAIN IN LATER MODULES.

DEVELOPMENT FEATURES

- SEPARATE COMPILATION: with CLAUSES
- VISIBILITY AND SCOPE
- PACKAGES
- STRONG TYPING

INSTRUCTOR NOTES

PROGRAMMING THE ADA MACHINE

- DON'T TRY TO PROGRAM THE "REAL" MACHINE.

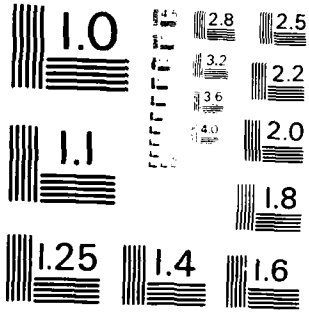
- THINK MORE ABOUT THE PROBLEM, LESS (AND LATER) ABOUT THE COMPUTER.
  - TYPES
  - MODULE STRUCTURE

- WORRY A LOT ABOUT READABILITY (HOW THE PROGRAM SOLVES THE PROBLEM, NOT HOW IT TICKLES THE COMPUTER).

AD-A141 848 INTRODUCTION TO ADA (TRADEMARK) A HIGHER ORDER LANGUAGE 4/4
L103 TEACHER'S GUIDE(U) SOFTECH INC WALTHAM MA MAY 84
DAAB07-83-C-K514

UNCLASSIFIED F/G 9/2 NL

END
11-84
DTIC



SUPPLEMENTARY

INFORMATION

DEPARTMENT OF THE ARMY
HEADQUARTERS US ARMY COMMUNICATIONS-ELECTRONICS COMMAND
AND FORT MONMOUTH
FORT MONMOUTH, NEW JERSEY 07703

REPLY TO ATTENTION OF:

15 OCT 1984

Center for Tactical Computer Systems

Ms. Madeline Crumbacker
Defense Tactical Information Center
Cameron Station
Alexandria, Virginia 22314

Dear Ms. Crumbacker:

As per phone conversation with Ms. Andrea Cappellini, CENTACS on 11 October 1984, a copyright statement has been omitted on documents sent to DTIC and NTIS. Enclosed please find the copyright statement (Encl 1) that must appear in the enclosed list of document (Encl 2). If you have any questions, please contact Ms. Cappellini at 201-544-4280.

Sincerely,

for JAMES E. SCHELL
Director, CENTACS